DOCUMENT RESUME

ED 284 678 PS 016 740

AUTHOR Carver, Sharon McCoy
TITLE Transfer of LOGO Debugging Skill: Analysis, Instruction, and Assessment.
SPONS AGENCY National Science Foundation, Washington, D.C.; Spencer Foundation, Chicago, Ill.
PUB DATE 22 Dec 86
GRANT NSF-MDR-8554464
NOTE 135p.; Ph.D. Dissertation, Carnegie-Mellon University.
PUB TYPE Dissertations/Theses - Doctoral Dissertations (041) -- Reports - Research/Technical (143)

EDRS PRICE MF01/PC06 Plus Postage.
DESCRIPTORS *Computer Assisted Instruction; Elementary Education; *Elementary School Students; Programing; Skill Development; *Task Analysis; Testing; *Transfer of Training
IDENTIFIERS *Debugging (Computers); *LOGO Programing Language

ABSTRACT

This dissertation seeks to determine the extent to which learning debugging in the context of LOGO programming improves children's debugging in other programming and nonprogramming contexts. The approach involves detailed task analysis of debugging (in the form of a computer simulation model), development of model-based instructional guidelines for teaching children debugging skills they do not learn "by discovery," and assessment of the debugging skills children are able to transfer to other programming and nonprogramming tasks. Twenty-two 8- to 11-year-old students took two 25-hour LOGO courses. Half of the students were taught debugging in the context of a LOGO graphics course first and then a LOGO list-processing course. The other half were taught debugging in the same two mini-courses, but in the reverse order. The performance of children taking tests in the first mini-course was compared with the performance of children taking the same tests in the second mini-course. Assessments of students' debugging skills revealed that the debugging strategies learned from explicit instruction in the first computer programming mini-course had a positive impact on children's debugging in both new programming and nonprogramming situations. (Author/PCB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
X This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

ED284678

Transfer of LOGO Debugging Skill: Analysis, Instruction, and Assessment

Sharon McCoy Carver

This thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Psychology.

Department or Psychology
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

22 December 1986

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
Sharon McCoy Carver
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

This research was supported in part by a National Science Foundation Graduate Fellowship and grants from the Spencer Foundation and the National Science Foundation (MDR-8554464).

PS 016740

BEST COPY AVAILABLE

Table of Contents

1. Seeking skill transfer from computer programming 3
1.1 The dream 4
1.1.1. A simple language with powerful constructs 5
1.1.2. The possibility of powerful ideas 7
1.2 The nightmare 8
1.3 The reality 11
1.3.1. Successful transfer of problem-solving skills 12
1.3.2. Realizing the dream of transfer from computer programming 14
2. Analyzing the components of debugging skill 15
2.1 A general model 15
2.2 A production system specification 17
2.2.1. Goals direct the solution 18
2.2.2. Heuristics narrow the search 18
2.2.3. Operators process information and produce behavior 19
2.2.4. Modeling solution strategies 21
2.3 Students' limited debugging skills 23
2.3.1. Debugging is a complex skill 25
2.3.2. Debugging skills require extra capacity 26
2.3.3. Debugging is not directly taught 28
2.4 Applications of the model 28
2.4.1. Designing instruction 28
2.4.2. Predicting and assessing transfer 29
3. Designing model-based instruction and assessment 31
3.1 A combination of transfer designs 31
3.1.1. A pre-test/post-test design 31
3.1.2. A savings design 33
3.2 The classroom and the classes 34
3.3 Instruction techniques 35
3.3.1. LOGO skills 36
3.3.2. Debugging skills 37
3.4 Data collection issues 39
3.4.1. Protocols 39
3.4.2. Partners 40
3.4.3. Prodding 40
3.5 Procedures for assessing skill acquisition 41
3.5.1. Programming tests 41
3.5.2. Debugging tests 42
3.5.3. Editing tests 43
3.6 Procedures for assessing debugging transfer 44
3.6.1. Similarity between debugging programs and debugging directions 45
3.6.2. Possible solution strategies 46
3.6.3. Transfer tests 47
4. Skill acquisition 49
4.1 Debugging skills 49
4.1.1. Achievement 51
4.1.2. Speed 52

4.1.3. Efficiency 52
4.1.4. Pre-search clue gathering 53
4.1.5. Faulty search 54
4.1.6. Incorrect changes 56
4.1.7. Experimenter help 56
4.1.8. Additional testing strategies 57
4.1.9. Reaction to debugging instruction 57
4.2 Programming skills 58
4.2.1. Achievement 58
4.2.2. Structured programming 59
4.2.3. Common errors and misconceptions 61
4.2.4. Debugging during programming 62
4.2.5. Independence 62
4.3 Editing skills 64
4.3.1. Speed 64
4.3.2. Efficiency 65
4.3.3. Incorrect changes 66
4.3.4. Experimenter help 66
4.3.5. Graphics versus list-processing 67
4.4 Acquisition Summary 67
5. Debugging skill transfer 69
5.1 Pre-search 69
5.2 Qualitative search analysis 69
5.3 Accuracy 71
5.4 Solution time 72
5.5 Checking 72
5.6 Transfer Summary 73
6. Conclusions 74
6.1 Support for the thesis 74
6.2 Possibilities for strengthening the evidence 77
6.3 Applying the findings and the approach 79
References 122
I. The GRAPES Production System for LOGO Debugging 129
I.1 Productions 129
I.2 Lisp Functions 162
I.3 Working Memory Elements 171
II. Traces of the Production System Debugging LOGO Programs 178
II.1 Low-Knowledge Graphics Debugging 178
II.2 Recovery from Debugging Errors 186
II.3 High-Knowledge Graphics Debugging 192
II.4 High-Knowledge Syntax Debugging 195
II.5 High-Knowledge List-Processing Debugging 199
II.6 More High-Knowledge List-Processing Debugging 203
III. Transcripts of Debugging Lessons 207
IV. Programming Tests: Plans and Experimenter Programs 235
V. Debugging Tests: Buggy Output and Programs 247
VI. Editing Tests: Marked Hard-Copy 261

VII. Transfer Tests 267
VIII. Standard Program Units and Structure 291

List of Figures

Figure 1:	Sample Graphics Programs	92
Figure 2:	Sample List-processing Programs	93
Figure 3:	The Goal Structure of the GRAPES Model	94
Figure 4:	A Sample Debugging Problem	95
Figure 5:	High Information Goal Tree	96
Figure 6:	Comparison of High and Low Information Goal Trees	97
Figure 7:	A Pre-test/Post-test Transfer Design.	98
Figure 8:	Patterns of Results Suggested by Alternative Hypotheses.	99
Figure 9:	A Savings Transfer Design.	100
Figure 10:	A Combined Pre-test/Post-test and Savings Design.	101
Figure 11:	The Model-Based Debugging Instruction	102
Figure 12:	Planned and Buggy Outcomes for Arranging Furniture	103
Figure 13:	A Sample Debugging Transcript	104
Figure 14:	Debugging Success	105
Figure 15:	Debugging Speed	106
Figure 16:	Debugging Efficiency	107
Figure 17:	Accuracy of Search Comments	108
Figure 18:	Amount of Pre-Search Comments	109
Figure 19:	Amount of Program Goal Achieved	110
Figure 20:	Amount of Program Structure	111
Figure 21:	Editing Speed	112
Figure 22:	Debugging Speed Minus Editing Speed	113
Figure 23:	Editing Efficiency	114
Figure 24:	Incorrect Edits	115
Figure 25:	Amount of Help Needed for Editing	116
Figure 26:	Reading Strategies on Transfer Tests	117
Figure 27:	Simulation Strategies on Transfer Tests	118
Figure 28:	Success for Debugging Directions	119
Figure 29:	Time to Debug Directions	120
Figure 30:	Use of the Checking Strategy	121

List of Tables

Table 1:	Discrepancy-Bug Mappings in the GRAPES Model	80
Table 2:	Example Trace of the GRAPES Model with High Information	81
Table 3:	Example Trace of the GRAPES Model with Low Information	83
Table 4:	Description of the Subjects	88
Table 5:	Sequence of Graphics Lessons	89
Table 6:	Sequence of List-processing Lessons	90
Table 7:	Buggy Directions for Arranging Furniture	91

Acknowledgements

Many people have helped me to survive graduate school and complete this thesis. My colleagues in the Department of Psychology have provided a challenging and enjoyable working environment. My family and friends have continually provided support and encouragement. And the people of the Crafton Heights U. P. Church have cared for me and made Pittsburgh my home.

Special thanks to:

- George Miller for convincing me to go to graduate school in the first place.
- David Klahr for treating me as a collaborator rather than a student and for sharing his interest and expertise in research with me.
- Yolanda Glasso, Yolanda Sweenie, Barbara Carlblom, and the upper elementary children at the Montessori Centre Academy for their enthusiastic participation in my experimental classroom.
- Bob Siegler, Catherine Sophian, Pat Carpenter, Lynne Reder, John Anderson, and Kevin Dunbar for offering constructive criticism of my research.
- Bonnie John, Becky Freeland, Mary Hegarty, and Kate McGilly for keeping me sane, nourished, and fit.
- Sandy Curry, Muriel Fleishman, and Lou Beckstrom for assisting me with the everyday details of graduate school.
- Cathie Raisbeck, Nancy Gilbert, and especially Sally Risinger for transcribing hundreds of hours of videotape.
- My husband, Dave, my friend, Carolanne, and my parents for enduring so much because they believed in me. To them, I dedicate this work.

Abstract

The goal of this dissertation is to determine the extent to which learning debugging in the context of LOGO programming improves children's debugging in other programming and non-programming contexts. The approach involves detailed task analysis of debugging (in the form of a computer simulation model), development of model-based instructional guidelines for teaching children debugging skills they do not learn "by discovery," and assessment of the debugging skills children are able to transfer to other programming and non-programming tasks. Twenty-two 8- to 11-year-old students took two 25 hour LOGO courses. Half of the students were taught debugging in the context of a LOGO graphics course first and then a LOGO list-processing course. The other half were taught debugging in the same two mini-courses but in the reverse order. Debugging skills were tested at three times during each mini-course. The performance of children taking tests in the first mini-course was compared with the performance of children taking the same tests in the second mini-course to reveal the transfer from one LOGO domain to the other. Debugging on non-programming tasks was assessed prior to the first mini-course, between mini-courses, and after the last mini-course to assess more remote transfer of debugging skills. Assessments of students' debugging skill revealed large savings from the first to the second mini-course. Students' increasing use of selective search strategies increased the accuracy, efficiency, and speed of their debugging. Corresponding improvements were demonstrated on a variety of tasks requiring debugging of non-computer directions. Children shifted from exhaustive to selective search strategies which increased the accuracy, efficiency, and speed with which they debugged written directions. Thus, the debugging strategies learned from explicit instruction in the the first computer programming mini-course had a positive impact on children's debugging in both new programming and non-programming situations.

1. Seeking skill transfer from computer programming

Transfer of training is the educator's dream but the researcher's nightmare. Clearly, teachers will be most effective when the learning of one instructional unit contributes to the learning of a subsequent unit or when a current lesson is facilitated by emphasizing relevant previous learning. At the same time, teachers hope that school lessons can be useful for solving "real world" problems. Essentially, educators strive to teach transferable skills because it would be impossible to teach all the knowledge students will need in their lifetimes. Despite the educator's dream, transfer is very difficult for researchers to demonstrate. Studies of human problem solving consistently find that experience with one problem rarely yields transfer to other problems even if they are similar. In the face of these negative results, could more distant transfer ever be expected? Or should educators abandon their dream?

The domain of LOGO computer programming is an interesting case in point because of the striking contrast between the grandiose claims for transfer of high-level thinking skills from programming experience and the largely negative results of LOGO transfer studies. This dissertation focuses on one central aspect of learning to program: *learning how to debug* faulty programs. This aspect has been identified as one of the "powerful ideas" that can generalize far beyond the programming context in which it is acquired (Papert, 1980). In more practical terms, debugging is a good candidate for special focus since failure to acquire good debugging skills could represent a "significant bottleneck to the development of programming competencies and whatever thinking skills may be fostered through high-level programming proficiencies."[1]

Transfer of debugging skill is expected because children discover in programming environments that 1) complex procedures can be constructed from subcomponents, 2) errors may derive from just a few buggy components, and 3) sources of error can be detected and corrected (debugged). In other environments, children generalize that 1) most of what they do is constructed from smaller components, 2) errors may derive from just a few buggy components, and 3) sources of error can be detected and corrected (debugged). Papert (1980) suggests the power of the debugging idea by saying: "Errors benefit us because

---

[1]This comment came from an insightful but anonymous reviewer.

they lead us to study what happened, to understand what went wrong, and, through understanding, to fix it. Experience with computer programming leads children more effectively than any other activity to 'believe in' debugging."

Unfortunately, this notion of transfer and the term "debugging" itself are ambiguous; this makes it difficult to evaluate claims for the cognitive consequences of learning debugging in a LOGO environment. Debugging can be interpreted along a wide spectrum, ranging from an all-encompassing notion of self-improvement, to a more restricted view of eliminating faulty components in physical or mental procedures[2], to a constrained definition that is closest to its origins in computer programming: it is what one does to get a malfunctioning (buggy) computer program to work correctly. The potential for transfer of debugging skills is also difficult to interpret. The goal of this dissertation is to use a detailed performance theory as the basis for determining the extent to which learning debugging in a LOGO context improves children's debugging in other programming and non-programming domains.

In this chapter, I will describe the LOGO computer programming language and the high-level thinking skills educators dream that students acquire and transfer to other domains. Then I will briefly review the literature on the reality of transfer from LOGO programming experience and propose several reasons for the primarily negative results of the LOGO transfer studies. Finally, I will develop an approach for facilitating and assessing transfer in the LOGO domain based on a review of several transfer success stories from the adult problem-solving literature.

1.1 The dream

Proponents of LOGO claim that programming experience can expand children's intellectual power (Papert, 1972, 1980). They claim LOGO becomes a tool for life: once it becomes a mental model for the child, his thinking processes become more conscious and he can master concepts previously thought too abstract. The key element of LOGO is said to be that powerful ideas are embedded in a simple language. The next two sections will describe the powerful constructs of this simple language and then enumerate the powerful ideas children may be able to learn from experience with this domain.

---

[2]For example, juggling is one of Papert's favorite domains for demonstrating learning via debugging. Also Brown & Burton (1978) have convincingly demonstrated the need for debugging in children's acquisition of arithmetic procedures.

1.1.1. A simple language with powerful constructs

At first glance, LOGO appears to be a simplistic programming language designed solely to enable young children with little training to create interesting graphic effects. Yet beneath its facade of simplicity, LOGO is a powerful programming language which allows subprocedures, variables, and recursion in graphics as well as list-processing. The following discussion will demonstrate the simple yet powerful nature of LOGO by briefly introducing the graphics and list-processing domains.

In LOGO graphics, the user directs the movement of the cursor around the screen using four basic commands: FD (forward), BK (back), LT (left turn), and RT (right turn). Each of these commands requires a numerical argument to indicate the distance to move (for FD and BK) or the number of degrees to turn (for LT and RT). In addition, PU (penup) and PD (pendown) control the position of an imaginary pen; when the pen is down, any cursor movement leaves a trace on the screen. With these six, short, semantically meaningful commands, young children can create pleasing designs in LOGO's interactive mode.

However, children can begin to utilize the power of LOGO by using its simple screen editor to write and revise programs. These programs can be called interactively or from within other programs. LOGO also has primitives to direct the flow of control; these include REPEAT n [list of commands], which repeats the list of commands n times; IF <conditional> THEN <commands> ELSE <commands>, which is a basic conditional statement; and STOP, which stops the execution of the current program and returns control to the calling procedure.

The example programs in Figure 1 demonstrate the basic graphics capabilities of LOGO. The procedure definitions are listed on the left. For easier reading, some commands are indented and arranged on separate lines. The interactive calls and outcomes of the procedures are on the right. The starting position of the turtle is indicated by an arrow. In Figure 1a, the REPEAT statement is used to write programs to draw a diamond and a curve. The leaf program is written using a repeated curve. These programs are combined to make a program to draw a flower. Each of the programs includes a variable (:D, :C, :B, and :A) so that one can draw flowers of various sizes. The first line of commands draws the leaves and the stem portions below and between the leaves. The second line draws the remainder of the stem, and the third line uses a REPEAT statement to draw ten diamonds at equally spaced orientations. Figure 1b illustrates how a recursive procedure can generate a row of flowers of decreasing size.

Insert Figure 1 about here

There is more to LOGO than graphics, however. The list-processing capabilities of LOGO are similar to LISP, the artificial intelligence language on which it is based. The user can PRINT information to the computer screen, MAKE global variable assignments, and READ keyboard input. OUTPUT makes the result of a procedure available to a calling procedure. LOGO also has commands to separate and join text; for example, FIRST takes the first element of a word or list and BUTFIRST takes the rest. WORD combines its arguments into a LOGO word and SENTENCE combines its arguments into a LOGO list. Here again, the commands are designed to be semantically meaningful. However, the syntax for the list-processing commands is more complicated than for the graphics commands because of the punctuation necessary to distinguish commands from variables from text.

The same powerful constructs described above can be used for list-processing. The example programs in Figure 2 demonstrate the basic list-processing capabilities of LOGO. Here, the procedure definitions are listed above the interactive calls and output. In Figure 2a, the PIGGY program translates an input word into piglatin by combining (WORD) all but the first letter (BUTFIRST), then the first letter (FIRST), and finally the "ay" ending. OUTPUT is used to pass the resulting word to another command, in this case, PRINT. Figure 2b illustrates the use of recursion to translate an entire sentence into piglatin. Finally, Figure 2c shows a program written to make the other two programs accessible to a naive user. It asks for a sentence to translate, does the translation, and offers an option to continue. The user's answer is set equal to the global variable Y and is then tested by the conditional in the next line. A "no" answer yields a farewell statement; any other answer causes the PIGLATIN program to run again.

Insert Figure 2 about here

As the graphics and list-processing examples demonstrate, LOGO offers an easy introduction to programming as well as opportunities to develop advanced programming skills. However, the great expectation is that children will learn thinking skills in addition to programming skills. Pea & Kurland (1984) comment on the widespread belief that

"...through learning to program, children are learning much more than

> programming, far more than programming 'facts'. It is said that children will acquire powerfully general higher cognitive skills such as planning abilities, problem-solving heuristics, and reflectiveness on the revisionary character of the problem solving process itself. This belief, although new in its application to this domain, is an old idea in a new costume which has been worn often before. In its common extreme form, it is based on an assumption about learning - that spontaneous experience with a powerful symbolic system will have beneficial cognitive consequences, especially for higher order cognitive skills. Similar arguments have been offered in centuries past for mathematics, logic, writing systems, and Latin (e.g., see Bruner, 1966; Cole & Griffin, 1980; Goody, 1977; Olson, 1976; Ong, 1982; Vygotsky, 1978)."

1.1.2. The possibility of powerful ideas

The "powerful ideas" children might develop as a result of experience with LOGO have been specified in slightly different ways by several researchers since they were introduced by Seymour Papert (1972, 1980).

P.H. Winston (1977), for example views the possible powerful ideas primarily in terms of programming concepts more broadly applied. He suggests applying the idea of state variables such as the turtle's orientation and position to the temperature in a room, applying the idea of control variables like the move and turn commands given to the turtle to other control variables such as force, applying the idea of subgoaling to other problems where a divide and conquer strategy is effective, and applying the idea of debugging to other situations where the initial solution is unlikely to be perfect but where the primary parts are basically correct.

In contrast, Feurzeig et al. (from Pea and Kurland, 1984) suggest that programming changes thought as a result of the thinking skills employed, not just the specific programming ideas a programmer must use. For example, rigorous thinking and precise expression will develop because the computer requires programmers to use such skills. Facility with general heuristics such as planning, analogy, and problem decomposition will increase since programming provides models of them. Debugging will improve because it is central to the interactive nature of programming. Self-consciousness and literacy about problem-solving processes will be heightened since programming provides a vocabulary for discussing these processes explicitly. In addition, general concepts of procedures, variables, and hierarchy will develop simply because they are ideas encouraged in programming. Also, recognition of multiple correct solutions, each with their own benefits, instead of a single best answer should develop since programming distinguishes process from product in that way.

Linn and Fisher (1983) describe powerful ideas arising from styles of interaction with the computer. The computer's interactive feedback should yield greater cognitive activity. The precision required by the interpreter should yield complete specifications of ideas. The consistency of feedback is beneficial to the student; there is no ambiguity and no prejudice. Linn and Fisher also suggest that the challenge of computer programming leads to greater motivation and that the possibility of multiple solutions leads to divergent thinking.

Finally, Clements and Gullo (1984) discuss powerful ideas as resulting from specific learning activities encountered in a LOGO environment. They suggest that divergent thinking should improve as a result of students' inventing, constructing, and modifying their own projects. Since students reflect on their own thinking processes, they should make metacognitive advances. Considering errors and fixes should increase reflectivity. As abstract ideas become concrete in the LOGO microworld, cognitive development should be accelerated. The experience of giving spatial commands to the turtle should improve students' perspective-taking ability when giving directions.

In theory, other environments could be equally good contexts for learning debugging and other high-level thinking skills. The advantage of LOGO over other environments is that it is a microworld with a limited set of precisely specified units which behave in a precisely specified manner. The LOGO language provides an external representation at an appropriate level of access for elementary school children. Such a simplified context may make debugging more comprehensible than in the "real" world of ill-specified units and unpredictable behavior. The dream is that, once learned, thinking skills may be generalizable beyond the LOGO microworld.

1.2 The nightmare

The transfer studies of LOGO are about as diverse as the interpretations of what powerful ideas are available for learning, and the results of these evaluations seem contradictory. Some researchers claim to demonstrate transfer, some fail to demonstrate transfer, and others get mixed results even within the same study. Gorman and Bourne (1983) and Degelman et al. (1986) demonstrate transfer from LOGO to a rule-learning task. Brown and Rood (1984) showed that LOGO and BASIC students improved in problem-solving ability, self-esteem, and internalized locus of control. Schwartz et al. (1984) report that LOGO students increase motivation and cognition scores more than a control group. They also show that the LOGO group's math anxiety increases less and confidence decreases less than the

control group. But Pea (1983) failed to show transfer of planning skills, and McGilly et al. (1984) found that LOGO students demonstrated no better skills in procedurality and debugging than students who had no LOGO experience. At the same time, Clements and Gullo (1984) and Clements (1985) reported that LOGO experience improved divergent thinking, reflectivity, direction giving, and metacognition, but not cognitive development. Similarly, Garlick (1984) found improvements in school test scores, but not in spatial relations or combinatorial thinking after LOGO experience. Mohamed (1985) showed that LOGO students improved in spatial ability but not quantitative ability, ability to synthesize, or analytic cognitive style.

This sampling does not exhaust the list of LOGO transfer studies (especially since many unsuccessful studies are undoubtedly not a part of the published or circulated literature); however, it is representative enough for the purposes of this discussion. This section will attempt to explain the nightmarish results in terms of the basic requirements for successful transfer. The next section will then attempt to focus on the real possibility for achieving transfer, based on the literature describing transfer of adult problem-solving skills.

By definition, transfer requires learning in the initial context and then exposure to a second context in which the learned knowledge and skills are relevant. The literature on LOGO transfer effects reports mixed results because of variation in the extent to which these two requirements are satisfied. In the LOGO domain, learning and transfer are usually studied in isolation, as if they were independent. Researchers interested in detailed accounts of learning frequently focus on extremely simple, low-level skills acquired over a few hours at most (see Roberts, 1984; Cuneo, 1985; and Campbell et al, 1985). Several researchers have examined the learning of higher level skills: for example, Kurland and Pea (1983) studied children's mental models of recursion. Mawby (1984) evaluated students' understanding of variables and control structure, and McBride (1985) has begun to analyze LOGO programmers developing strategies and knowledge. However, these studies of high-level skills are not coordinated with studies of transfer, perhaps because they reveal students' lack of significant high-level skill acquisition. On the other hand, researchers interested in LOGO transfer frequently fail to document that students have improved on the desired skills within the LOGO domain prior to evaluating transfer. Of the ten transfer studies listed above, only three included assessments of learning (Pea and Kurland, 1983; McGilly, 1984; and Garlick, 1984). Therefore, transfer of programming skill on the other studies was assessed without a prior determination that the skill was ever acquired in the

first place. Without an understanding of what LOGO students have actually learned (or not learned, as is often the case), transfer results are difficult to interpret. Positive results may be due to motivational factors, and negative results may only indicate that the required LOGO skills were never learned.

In addition to neglecting to document learning, most researchers studying transfer from LOGO fail to carefully consider the skills they expect students to learn or the skills required by their transfer tasks, so there is often a mismatch between the transfer test and the training. Though there is considerable variety in choice of transfer tests, positive transfer results in the studies listed above tend to be on tasks involving skills with figures similar to those used in LOGO, not high-level thinking skills. The school tests used in Garlick's (1984) transfer test involved questions about angles, distances, and similar figures. Clements and Gullo (1984) used only the figural subtask of the Torrance test of creative thinking and measured reflectivity using the Matching Familiar Figures Test. LOGO students only improved more on the spatial part of the Developing Cognitive Abilities Test in Mohamed's (1985) study. The skill similarity occasionally depends on the particular version of LOGO used. Skills necessary for rule-learning tasks may be learned better in TI-LOGO, used by Gorman and Bourne (1983), since it offers experience with animated sprites each of which has many independent attributes including color, number, heading, speed, and shape.

On the other hand, negative results occur on tasks whose relation to LOGO experience has not been specified, as well as usually not being documented. For example, Clements and Gullo (1984) sought transfer to classification and seriation tests of cognitive development and to metacognitive tasks (see also Clements, 1985), and Mohamed (1985) expected improvement in ability to synthesize and analyze. Neither research group specified what relevant LOGO skills students had actually learned. Pea and Kurland (1983) expected transfer of planning skills and Garlick (1984) expected transfer of combinatorial thinking skills. Both assessed learning of LOGO content and general programming skills, but neither specified nor assessed learning of the skills which were expected to transfer. McGilly et al. (1984) did document improvements in procedurality and debugging skills prior to assessing transfer of those skills; however, they failed to specify how the learned skills were relevant to the transfer tasks (i.e., Tower of Hanoi for assessing procedurality and Mastermind for assessing debugging).

In addition to the problems of not documenting learning and not specifying the skills to be

learned and transfered, there are many other methodological problems with most of the LOGO studies, so their results are questionable. For example, in many of the transfer studies, performance of different treatment groups was compared on a post-test without having any pre-test comparison (e.g., Degelman,1986; Gorman and Bourne,1983; Clements and Gullo, 1984; and Clements, 1985). Some which did include a pre-test comparison did not use random assignment and had treatment groups with higher pre-test scores than the control groups (e.g., Schwartz et al., 1984 and Garlick, 1984). Others had no control group at all (Gorman and Bourne,1983 and Brown and Rood, 1984). Finally, many of the reported transfer effects were actually small differences both in terms of the absolute numbers and in terms of the percentage improvement. (Schwartz et al. (1984) is the most striking example.)

The focus of this description has been on the principle that studies failing to demonstrate transfer from LOGO cannot be offered as negative evidence of the possible cognitive consequences of learning programming unless they first demonstrate that students actually learned the potentially transferable skills and then choose a transfer task on which those particular skills are useful, though there are problems in determining the levels of usefulness. Likewise, studies that successfully demonstrate transfer cannot be offered as positive evidence unless they also demonstrate that the effect is a result of students learning relevant skills during the LOGO experience.

Researchers in other domains have paid careful attention to assuring learning and specifying relevant skills for transfer. The next section will summarize their results and the guidelines they offer for facilitating transferable learning and choosing appropriate transfer tasks. These guidelines will be used to design a transfer study of LOGO debugging skills.

1.3 The reality

The literature on transfer of problem-solving skills has shown that when learning can be assured, transfer is likely. Furthermore, researchers have begun to specify what must be learned for transfer to occur and to suggest ways to facilitate learning. Also, the transfer literature has shown that transfer is better to tasks for which the relevance of the learned skills can be recognized. Most importantly, these researchers are developing ways to decide what is a relevant transfer task and what is not.

1.3.1. Successful transfer of problem-solving skills

Bassok and Holyoak (1986) suggest that many of the transfer failures can be attributed to insufficient learning since few researchers provide more than one trial training. In fact, Smith (1986) demonstrated that the provision of multiple training trials increased learning and transfer scores in a study of transfer between Tower of Hanoi isomorphs.

However, Bassok and Holyoak (1986) also suggest that overlearning is not sufficient to produce transfer. They view the difficulty of transfer as recognizing abstract structural commonalities despite surface differences. They claim that the acquisition of general schemas for problem types improves transfer since recognition of a novel instance is more likely. They suggest that the provision of multiple examples is crucial to induction of abstract schemas. Gick and Holyoak (1983) showed that transfer could be enhanced on isomorphic problems by giving solvers a direct hint to apply the previously used solution to the new problem and that even greater enhancement resulted from requesting that solvers describe the similarity between problems (and presumably abstract the relevant schema in the process).

Bassok and Holyoak (1986) compared transfer from algebra's arithmetic-progression problems to physics' constant-acceleration problems and vice versa. They hypothesized and found that there was more transfer from algebra to physics than in the other direction because the wider variety of arithmetic-progression problems yielded a more abstract schema which the subsequent physics problems were more likely to match. They described the general schema as production rules with abstract variables that can be readily matched by components of problems with the same structure. They concluded that training will produce transfer to structurally identical problems to the extent that the "abstract training in a problem schema is combined with practice in solving diverse examples." Bassok and Holyoak (1986) also claimed that another facilitating factor in the physics and algebra domains was that students were taught the skills directly.

Kotovsky, Hayes, and Simon (1985) claim that the amount of learning is also related to the source problem difficulty. The processing load determines how much information subjects learn on a first problem that they can then transfer to a second problem. They showed that increased difficulty on Tower of Hanoi isomorphs decreased the amount of transfer. They attributed the decrease to a limitation of working memory capacity on the acquisition of transferable information.

Even if abstract learning does take place in one domain, transfer to a second domain depends on the relevance of the learned skills to the new doamin. The original form of this idea was Thorndike's (1913) identical elements theory. He suggested that learning of one stimulus response link would only affect other such links if some of the factors were identical elements. Kieras and Bovair (1985) represented the elements of related procedures as production rules and demonstrated that there was more transfer between procedures when they had more identical rules. However, identical elements are difficult to identify and count. Other studies (mentioned above) show that it is possible to transfer without identical elements as long as the skill is represented abstractly enough, or production rules are general enough, that the relevance of the previously learned skills is apparent.

Singley and Anderson (1985) agree that transfer can be predicted by the degree of overlap between tasks. They showed that learning a second text editing system took less time, fewer keystrokes, and fewer errors despite a higher typing rate than learning the first. However, they attributed this savings to transfer of the high-level goal structure and the conceptual mappings between text-editing commands and their actions.

Dalbey and Linn's (1984) research demonstrates the application of this principle in the domain of children's programming. They found more transfer to "robot tests" from "Spider World," a LOGO-like graphics microworld, than from either BASIC or Type Attack. Though the commands are not identical in Spider World and the robot test, the actions of the commands and the goals of the task were quite similar. BASIC and Type Attack did not have this similarity with the robot tests so the lack of transfer in these conditions was not surprising.

All of these studies highlight the need for transfer assessments to be based on detailed understanding of the skills learned in the training phase and required in the transfer phase. Claims have been made that students will learn more transferable skills when their learning is abstract and when the memory load is low. These principles were incorporated into the debugging instruction provided to the students in the dissertation study to maximize their potential for learning transferable skills.

1.3.2. Realizing the dream of transfer from computer programming

Newell and Simon (1972) suggested a threefold research agenda for studying human problem solving that is consistent with the findings reported above: develop a detailed performance theory, then address issues of learning, and finally address issues of transfer. Similarly, Anderson (1987) stresses that his design of intelligent computer tutors is based on the premise that "one needs to have a cognitive process model of the student if one is going to be able to effectively tutor the student." Failure to base both instruction and assessment on a performance theory has led to the distressing mixture of results in the LOGO literature. Researchers in the other problem-solving domains are attempting to follow this agenda and are documenting learning and/or transfer. A search for transfer of skills from programming experience must follow a clear statement of what these skills are and an assessment of how well they are acquired during LOGO instruction.

My research agenda, therefore, includes three phases:

- Detailed production system analysis of debugging skills in the LOGO domain,
- Direct instruction of those skills in two domains (to facilitate abstraction) with external support to reduce the working memory demands, and
- Assessment of skill acquisition in the base domain and transfer to domains requiring similar skills.

Anderson et al. (1985) have reported positive results from using a similar approach to design and assess the impact of computer tutors for teaching both Lisp programming and geometry skills. In contrast, my approach will focus on designing principled instruction to be used by a human teacher in a typical classroom.

The rest of this thesis is organized as follows. Chapter 2 describes a detailed task analysis to model good debugging skill (in the form of a computer simulation) and summarizes the results of a pilot study designed to establish which of the model's skills students learn in a "typical" LOGO course (i.e., with no explicit instruction in debugging). Chapter 3 describes the experimental design and the model-based instruction and assessment techniques. Chapter 4 details the students' skill level in one LOGO mini-course and the savings observed when they moved into a second LOGO mini-course. Chapter 5 discusses the transfer of debugging skills from LOGO to non-programming domains. Finally, Chapter 6 summarizes the findings and suggests ways to strengthen and apply them.

2. Analyzing the components of debugging skill

The debugging instruction and transfer assessments discussed in this dissertation were based on a detailed task analysis of LOGO debugging skills. The analysis was intended to capture, in the form of a concrete model, the decision processes, knowledge, and sub-skills necessary for efficient debugging of LOGO graphics and list-processing programs with one or more semantic and/or syntactic bugs.

The following sections describe the model and its applications for instruction and assessment. First, Section 2.1 is a general characterization of the steps in the model's debugging process. Next, Section 2.2 is a description of the actual production system implementation of the model along with examples of it debugging faulty LOGO programs. Section 2.3 is a report of a pilot study designed to determine which components of the model students learn in a typical LOGO course. The chapter will conclude in Section 2.4 with a discussion of designing instruction and assessment in accordance with the model as well as predicting transfer from debugging training in the context of LOGO programming.

2.1 A general model

The assumed debugging situation is one in which the model has access to the program plan (the desired outcome), the buggy program, the output that the buggy program produces, and knowledge about LOGO. In the following analysis, we distinguish between the *discrepancy* and the *bug*. The former refers to the difference between the program plan and the program output. The latter refers to the erroneous component of the program that caused the discrepancy. The goal of the debugging process is to detect and correct the discrepancy-causing bug. For example, if the goal drawing corresponding to Figure 1a was a taller flower, then the discrepancy between the goal drawing and the program output would be described in terms of the difference in size. The bug that caused the discrepancy would be the * 15 in the FD :D * 15 command that draws the stem.

According to the model, there are five phases to the debugging process. The first phase establishes four subgoals that, when completed, reassert the top goal. The five phases are:

1. Program Evaluation. Run the program. Compare the program plan and the program output. If they do not match perfectly, then identify the bug, represent the program, locate the bug, and correct the bug.

2. Bug Identification. Generate a description of the discrepancy between the program plan and the program output. Based on the discrepancy description, propose specific types of bugs that might be responsible for the discrepancy. Where multiple possibilities exist, do further discrepancy description and bug proposal. When only one possibility remains, examine the program output to identify the specific bug.

    - In its purest form, the discrepancy description makes no reference to the fact that the faulty output is program-generated. That is, the discrepancies are characterized entirely in terms of their static features.[3] Table 1 lists the most common types of discrepancy encountered when debugging LOGO graphics and list-processing programs. The quotations presented in the second column are representative comments from children in this study about the type of discrepancy shown in the first column. Note that one possible outcome of the bug identification step is knowing that the plan and output are not identical but being unable to describe the mismatch. However, in the case of syntax errors, the error message always provides a description of the discrepancy for the user (though the user may ignore it).

    - Given the description of the discrepancy, the model makes inferences about which specific program components are capable of generating that type of discrepancy. The third column in Table 1 suggests some of the possible mappings. For example, if the discrepancy is spread, then it is likely to be caused by turning the wrong angle or moving the wrong distance. In addition to proposing these general types of programming errors, the model has a set of rules which propose further discrepancy description to discriminate between multiple possibilities. When only one possibility remains, the model examines the program output to determine the specific bug. The model may need to cycle through discrepancy description and bug proposal several times before a specific program command is identified as the bug (see the fourth column in Table 1). However, the result of this complex processing is a narrower search for the bug (e.g., "right here - it shouldn't be left 90 - it should be right 90 I think").

3. Program Representation. Represent the structure of the program to investigate the probable location of the buggy command in the program listing.

    - Knowledge of the program's structure may be the result of having written the program or of assuming that programs for certain types of plans will be structured in characteristic ways. For example, the model may be given knowledge that the program has a repeat structure because the user wrote the program or because the user observes that a picture is composed of several identical figures (typically programmed using a REPEAT statement). Knowing that the bug is located within a REPEAT

---

[3] It is possible that discrepancy descriptions might include temporal information, because in our procedure, the child watches as the program's output is dynamically generated. On the computer we used, Figure 1a would take about 5 seconds to draw. Also, for list-processing output, the temporal order of different portions of the output is preserved by the listing on the screen.

statement narrows the search in the program listing. In the case of syntax errors, the error message gives the user information about which procedure contains the bug.

4. Bug Location. Using the cues gathered in the last two phases, examine the program in order to locate the alleged bug.

    - The efficiency of the bug location process depends on the outcome of the bug identification and program representation processes. At best, the model searches for a perfectly specified bug (both the buggy command and its arguments are specified) in a highly constrained set of possible bug locations. At worst, the model must perform a step-by-step examination of the program because it has no knowledge of the bug's identity and no cues about its location.

5. Bug Correction. Examine the program plan to determine the appropriate correction. Replace the bug with the correction in the program listing and then reevaluate the program.

    - This reevalutation is slightly different from the initial test in that the model knows a change has just been made. It first determines whether the correction fixed the original problem. If the correction worked, the model will determine whether there are any more bugs to fix; otherwise, it will debug the correction before proceeding.

Insert Table 1 about here

2.2 A production system specification

In order to specify the model unambiguously and to demonstrate its sufficiency for debugging LOGO programs, it was implemented in GRAPES, a goal-restricted production system (Sauers and Farrell, 1982). The GRAPES model consists of a set of rules, called productions, which specify the action to be taken if certain conditions exist. The conditions include the goal the model is trying to achieve and the information currently available in working memory (the set of known facts). A production is selected and executed only when the appropriate conditions exist; thus the current state of the environment determines which actions will be performed. The actions include updating or adding to both working memory and goal memory.

The 84 productions in our GRAPES model represent our task analysis of debugging skills in the LOGO context. The model's goals represent the steps in the debugging process; the heuristics it employs represent general and specific search strategies used for efficient

debugging; and the operators it invokes represent sub-skills which are essential, but not central, to the debugging process (e.g., editing skills). The following sections will describe these three components of the model in detail. These descriptions will be followed by demonstrations of how the goals, heuristics, and operators work together to debug faulty LOGO programs.

2.2.1. Goals direct the solution

The debugging model's goal structure corresponds to the five phases described in the overview. A goal tree is shown in Figure 3. The system has a set of productions for each goal to represent the different responses a debugger would have to the same goal in different situations. The "situations" are represented by the current contents of the system's working memory. Productions with **test** and **evaluate** goals start the system and evaluate the success of each debugging attempt (i.e., the match between the program plan and the program output). The **describe** and **propose** goals correspond to the bug identification phase; they satisfy the productions that describe the discrepancy between the program plan and the program's buggy output and that propose possible bugs and ways to discriminate among them. **Represent** and **specify** correspond to the program representation phase; productions with these goals look for structural cues to the bug's location so that **find, interpret,** and **check** can actually isolate the bug using whatever cues they have about the bug's identity and location. Finally, the **change** and **replace** goals correspond to the bug correction phase; they fire productions that identify the appropriate correction and change the program listing accordingly. The full production system is presented in Appendix I.

Insert Figure 3 about here

2.2.2. Heuristics narrow the search

The system has two sets of debugging heuristics, one set for identifying the bug and one set for representing the location of the bug in the program. Using both sets of heuristics narrows the search for the bug substantially. Heuristics for identifying the bug correspond to the mappings between observed discrepancies and potential bugs (listed in Table 1). These heuristics are most useful when there is more than one type of bug which can lead to a particular type of discrepancy. In this case the heuristic includes information for distinguishing between them. For example, if the discrepancy has been initially identified as spread, then the model will request information about orientation because it has the

knowledge that discrepancies described as both spread and orientation must have been caused by an angle bug whereas those described only as spread discrepancies must have been caused by a distance bug.

Heuristics for locating the bug involve knowledge of program structure types. For example, if the program is identified as having subprogram structure, the model would ask for information about which subprogram was likely to contain the error and it would confine its search to that subprogram unless no bug could be located there. If no subprogram cue is available, the model will seek other structural cues, such as location within a REPEAT or IF statement or location after a particular command. For example, if the user can identify a correct command which was executed before the bug occurred, the model will use that command as a marker and begin its search after that command.

2.2.3. Operators process information and produce behavior

According to our model, the debugging process uses 11 operators, or sub-skills, to process information available to the system. These operators are called by productions when it is necessary to process information from one or more sources or to take specific actions. Four sources of information are available to the operators at all times: the program plan, the program output, the program listing, and knowledge of the programming language. In addition, an operator may use local information contained in a working memory element. Operators may also add information to working memory.

There are two classes of operators: a) those that correspond to inspection of the buggy output and/or the plan, and b) those that correspond to maneuvering in the LOGO environment. The latter set of operators are automatically executed by the model, but the former set are not. Instead, their operation is simulated by the user of the system. Essentially, the user compares the buggy output and the program plan for the system, inputting judgments of whether a program did what it should have, estimating angles and distances, reading error messages from the LOGO screen, etc.

A brief description of each operator follows. Working memory elements are presented in parentheses and italics are used for variables representing the system user's input. For example, the working memory element **(discrepancy** *response* **yes)** might actually be instantiated as **(discrepancy orientation yes)**.

- The MATCH operator is called to process information from the program plan and the program output and input its judgment about the match between the two. The system questions the user, "Did the outcome match the plan?," or "Did the simulation match the plan?," and expects a yes or no response. A yes response causes the element (match yes) to enter working memory; likewise, a no response yields the element (match no). If a command has just been changed, the system's query is, "Did the correction fix the problem?" and the resulting working memory element is (fix yes) or (fix no).

- CONTRAST is a discrepancy-description operator that processes information from the program plan and the program output. The system first asks about the type of discrepancy (graphics or lists, semantics or syntax). Then it asks a more focused question about the particular discrepancy and gives a list of possibilities. For example, the system asks, "What is the discrepancy between the plan and outcome?" and requests one of the following answers: orientation, size, spread, location, extent, or ? for graphic semantic discrepancies. A working memory element of the form (discrepancy *response* yes) is then added to working memory. CONTRAST can also indicate whether a specified discrepancy exists or not. If a working memory element such as (description must be about size) exists, then the system's query is, "Is size discrepant?" A yes or no response is expected and the resulting working memory element is of the form (discrepancy size *response*).

- EXAMINE is a bug proposal operator which seeks specific information about the bug using LOGO knowledge to guide the processing of information from the program outcome. The system asks the user a question such as "What is the discrepant angle on the outcome?" and labels the user's response as the bug, (the bug could be (RT (120))) for example.

- INTERPRET is a bug location operator which simulates the effect of the current command using LOGO knowledge. The system reminds the user of the function of a command, then asks whether that was the appropriate command to use, then asks whether each of the arguments is correct. These questions are calls to the MATCH operator described above.

- GENERATE is a bug correction operator which uses LOGO knowledge to determine what command would be necessary to accomplish the desired effect. The system asks the user, "What should the command *bug* have been?," or "What command should be inserted?," and creates a new working memory element from the user's response, (the correction is (RT 90)) for example.

- The other six operators, basically editing operators, are associated with physical action in the LOGO environment. The model automatically carries out these operations on its representation of the LOGO environment, while notifing the user that it is doing so. These operators include RUNning the program, ENTERing the editor to view the program listing, SKIPping to a particular location in the program listing, READing a command, DELETing a command, and INSERTing a command.

2.2.4. Modeling solution strategies

The model's solution to a debugging problem depends on the amount of information gathered about the debugging situation to guide the search for the bug and the accuracy of the input resulting from the user-simulated operators. In this section, we contrast two debugging examples with different amounts of knowledge. The more knowledge input to the model, the narrower the search for the bug. Appendix II provides additional examples to show how the model recovers from being given incorrect input and to demonstrate its flexibility in dealing with a wide variety of debugging situations.

This example is extracted from one of the actual graphics debugging tests used in the dissertation study. The desired outcome was for LOGO to draw a corn field (Figure 4a). The program's outcome was, however, discrepant from the plan (Figure 4b) because there was a bug in the program. The relevant portion of the test program is shown in Figure 4c. The two traces discussed in this section (Tables 2 and 3) differ in the amount of information about the bug's identity and location. The goal numbers in the discussion refer to these traces.

Insert Figure 4 about here

In the first trace (Table 2), we simulate a situation in which the debugger is a very knowledgeable user. The model is provided with a lot of information about both the discrepancy and the program. The information, provided in response to the operators, is marked by --> on the right-hand side of the trace. Here the user classifies the problem as graphics without an error message (goal-2) and then identifies the discrepancy type as "wrongpart" (goal-6) since the outcome has four ears of corn instead of four stalks. The model then proposes that the wrong subprogram has been called (goal-7). It asks the user the name of the wrong subprogram. The user responds that it is called CORN. This user's clever discrepancy description led to a very specific proposal of the bug; other descriptions, e.g. "missingpart", would have worked also, though perhaps not as efficiently.

When prompted about the subprogram structure of the program, the user responds with a ? (goal-3), indicating uncertainty about the structure and about whether the bug is in a substructure. The user then indicates that the bug is probably in a REPEAT statement (goals 8-9) since there are four identical figures. Knowing that the bug is in a REPEAT statement plus that it is probably a wrong subprogram call to CORN allows the model to

find the bug directly (goal-4). The model asks the user whether CORN should be changed, deleted, or have another command inserted before it (goal-5). The user indicates that a change is necessary so the model requests the user to input the correction (goal-11). It makes the correction and then instructs the user to rerun the program to check the correction (goals 12-14). The correction is accurate so the model asks whether there are any more problems (goal-15). Since there are not, debugging is complete. Figure 5 shows the goal tree generated during this debugging cycle.

Insert Table 2 about here

Insert Figure 5 about here

The second trace, in Table 3, illustrates the model's behavior when the user knows nothing about the discrepancy between the plan and the outcome and nothing about the structure of the program. The user runs the program and knows that a discrepancy exists (top-goal). When asked about the type of discrepancy, the user knows that it is a graphics problem without an error message (goal-2). The user responds to the specific type of graphics discrepancy with a ? so the model cannot propose the bug's identity (goals 6-7). When asked about the program's structure, the user again answers with a ? so the model tries to get information about REPEAT and IF statements and other commands that might be useful as markers (goals 3,8-10). Again the user responds negatively so the model must search for the bug by iterating through the program listing (goals 4,11-28). For each command, the model defines the command and asks the user whether it was the appropriate command to use. Then it asks the user to check each argument. The first command in the REPEAT statement is then determined by the user to be the bug (goal-29). The model asks whether a change, a deletion, or an insertion would be appropriate and the user decides to make a change (goal-5). The model requests a correction which, once it is input by the user, will be substituted into the program listing in the place of the bug (goals 30-31). Finally, the model directs the user to rerun the program to be sure the correction was accurate (goals 32-33). Since it was and no other discrepancies existed (goals 34-35), the debugging episode was complete.

Insert Table 3 about here

Figure 6 compares schematic versions of the goal trees for the two traces. The contrasts between the model's behavior in the high- and low-information situations are striking: the former required only 17 production firings, while the latter required 52. With respect to the complexity of the sub-goal tree, shown in Figure 6, we see the same kind of difference: 16 sub-goals vs 35. The high-information simulation represents the ideal case in which the bug is completely specified and its location is known. The system's goals and heuristics were used efficiently to narrow the search for the bug. In the low-information situation, little use is made of the describe, propose, represent, and specify goals so none of the heuristics for narrowing the debugger's search are used and debugging proceeds by brute force, one command at a time. Most of the extra production firings and sub-goals result from this serial search.

Insert Figure 6 about here

For the purpose of this simulation, we chose an example where the bug was close to the beginning of the program and assumed that the interpret operator correctly identified the bug. If this had not been the case, the difference between the two traces would have been even more striking because longer and/or repeated debugging cycles would have been necessary. Interested readers should peruse the traces in Appendix II to get a more global picture of how the model works.

2.3 Students' limited debugging skills

A pilot study (Carver and Klahr, 1986) used the formal task analysis of debugging as a context for assessing how much of the debugging skill specified by the model children actually learn in a guided discovery LOGO graphics environment. A guided discovery environment is one in which the instructor introduces new LOGO concepts and may give project ideas for trying them, but then the students are free to create projects of their own choice. The course was designed to assess the acquisition of both debugging skills, the model's goals, heuristics, and debugging operators (MATCH, CONTRAST, and EXAMINE), and conventional LOGO sub-skills (the model's general operators). Debugging skills were evaluated at three times during the LOGO course by asking students to describe the problem with the output of several programs and then asking them to view and debug the programs. We assessed the other operators assumed by the model at three times also. The ability to INTERPRET commands to predict the behavior they would cause was

assessed using a paper and pencil task; students were given programs and asked to draw their outcomes. We used a turtle target game to assess students' ability to GENERATE commands to cause specific behavior: students had to give the turtle one turn command and one move command to make it reach a target. The ability to maneuver within the LOGO programming environment (RUN, ENTER, SKIP, READ, DELETE, and INSERT) was tested using a program editing task; students were given a hard-copy of a program with changes marked on it and they were asked to do the editing.

Nine children (5 females and 4 males) ranging in age from 7;1 to 8;9 participated in the study. They were recruited from the communities surrounding Carnegie-Mellon University by advertising a free LOGO course. None of the children knew any computer programming languages, but most of them had limited experience either with computer games or with programmed instruction. Eight of the subjects came in pairs and one came individually to 12 two-hour LOGO classes over a three week period during the summer. All lessons were taught by a 24-year-old experimenter who was an experienced LOGO instructor. The instruction used an APPLE*IIe computer with Terrapin LOGO. Skills in command interpretation, command generation, maneuvering within the LOGO environment, and debugging were tested three times during the course.

After 24 hours of LOGO experience, which is as much or more experience than was provided in 9 of the 10 studies mentioned in Chapter 1 (Clements, 1985), our subjects had not developed effective debugging strategies. In fact, they rarely debugged programs when they were not required to do so. Figure 3 depicts the general results in terms of the proposed debugging model: the boxes around the goal names indicate the parts of the model with which the children had difficulty. In the **test** phase, children were able to MATCH the goal drawing and the program output to determine whether or not they were the same. However, they may have found debugging tedious because, according to our model, they lacked knowledge of the discrepancy-bug mappings and heuristics for locating the bug in the program listing. When forced to comment on the nature of the problem, they demonstrated limited ability to **describe** the discrepancy. Also, their **propose** phase was not useful because of their inability to discriminate between potential bugs and their poor EXAMINE operator. In addition to this restriction on debugging, the children had few cues to **represent** the program (since they had not written it). Even when clues had been mentioned, they were seldom used to narrow the search for the bug, so children had to **find** and **change** the bug using step-by-step examination of the program. Despite good

skills in maneuvering within the LOGO environment (RUN, ENTER, SKIP, etc.), the serial search was tedious because of the children's poorly developed skills at INTERPRETing commands. However, children were generally good at GENERATing the command to correct the bug once it had been located. Though the children in Carver and Klahr's pilot study did learn some LOGO skills, they did not learn the goal structure, the heuristics, or the key operators that the model uses for effective debugging. Also, in their own debugging, they tended to skip the phases before looking at the program so they were usually left to serial search when they debugged at all.

The debugging model assumes that the child has already made the decision to debug a program, rather than to simply abandon it and start over. However, Carver and Klahr's (1986) pilot study as well as many educators' informal observation (Papert, 1980) is that children prefer to restart rather than to debug. We believe that in most cases children don't debug because they don't know how to. There are three reasons why they fail to acquire this skill: a) Debugging is a complex skill; b) it requires extra memory capacity; and c) it is rarely taught directly. Below, we elaborate each of these stumbling blocks in relation to the model and to the relevant literature.

2.3.1. Debugging is a complex skill

Debugging's complexity is obvious from the large number of **goals**, **heuristics**, and operators necessary to describe our model. The bug identification and **bug** location productions represent a minimal set of heuristics for finding bugs; without these search shortcuts, the model's debugging is laborious. Well-developed operators are essential for accurately comparing the actual output with the goal output and for interacting with the LOGO system; without such operators, the model makes frequent errors and requires many cycles to correctly debug a program.

Research on adult programming skills has shown similar difficulties among novices. Studies with adults have shown that novice programmers fail to use the goal structure of a program as an aid to bug isolation. For example, Jeffries' (1982) study of expert and novice Pascal programmers showed that growing expertise involved developing a hierarchical representation of programs. (She also found that experts had accummulated a set of familiar patterns that they used to relate flaws in the output to potential bugs. Our model represents such knowledge in the **propose** productions.) Spohrer, Soloway, and Pope (1985) make the interesting suggestion that failure to maintain the appropriate goal hierarchy ("goal

dropout" and "merged goals") is a common source of bugs. Atwood and Ramsey (1978) found that debugging was difficult for adult novice FORTRAN programmers because they lacked useful heuristics for seeking cues from faulty output which could narrow their search for the bug.

Also, Littman (1986) has shown that even expert programmers do not use an effective search-narrowing strategy ("as-needed," in Littman's terms) when attempting to modify large programs. Rather, the experts who used a brute force strategy ("systematic," in Littman's terms) succeeded more often than those using the as-needed strategy. The success of the systematic strategy depends heavily on a good interpret operator and a program of manageable size. Good as-needed strategies, similar to the cue-directed search the debugging model uses, would be necessary for large software projects. However, the experts who did use the as-needed search in Littman's study did not demonstrate good strategies; they had difficulty determining what was "needed."

Although goal structures play an important role in adult programming, a hierarchical conceptualization of the solution to a programming problem is very rare at the low level of programming skill typically reached by children, which is perhaps why they do poorly. For example, in Pea's (1983) study, children were able to debug syntax errors effectively but were not able to locate semantic bugs such as misordered commands. Part of their difficulty was a result of their tendency to approach programs as long chains of direct commands rather than as hierarchical structures. Nevertheless, because we wanted our model to represent an efficient debugger, we did include the capacity to effectively use knowledge about the goal structure of the buggy program (in addition to a last-resort brute force strategy). If the responses to the represent operators indicate that the program has a specific structure and that the bug appears to be local to a particular component of that structure, then the model immediately constrains its search to that location.

2.3.2. Debugging skills require extra capacity

Development and use of debugging skills requires memory capacity sufficient to keep track of available cues. Even though most of the productions in our model require only two or three working memory elements as conditions, this is more than novices can handle in addition to the already heavy memory load imposed by learning to program. Also, the context in which students typically experience a need for debugging is one in which their attention is directed toward the problem at hand: getting the program right. This focus

leaves them little capacity to learn about the debugging process itself. According to Kotovsky et al. (1985), this minimal available capacity would result in minimal learning and therefore minimal transfer. Anderson and Jeffries (1985) suggest that many of the errors adult novices make in computer programming were the result of working memory failure. Children may be even more susceptible to this difficulty.

In fact, Pea and Kurland (1984) suggest a a multi-stage characterization of the acquisition of programming skills that emphasizes the capacity issue. The beginner is simply a "code generator" who focuses on individual commands rather than developing a structured program. Next, the student begins to think in terms of higher level units, becoming a "program generator" who can create and debug complex programs. Finally, the student becomes sufficiently familiar with the language that he can distance himself from the coding processes to consider the general problem-solving aspects of programming such as elegance, efficiency, and optimization; he has become a "software developer." Only at this level can he deal with high-level thinking skills to the extent that transfer would be possible. Much evidence suggests that even after 20 to 30 hours of instruction, most children are barely out of the first stage (Carver and Klahr, 1986; Pea and Kurland, 1983; Mawby, 1984); they are still struggling to acquire the basic LOGO operators. Also, Jenkins (1986) showed that even a small sample of LOGO teachers failed to debug complex LOGO programs.

Focusing on the complexity of debugging and the need for extra capacity to learn it has shaped the agenda for research on novice programming. Most researchers study novice difficulties so that instruction can be improved and students can reach the high-level skills more quickly. Mayer (1981) viewed the novice's problem as one of concept meaninglessness, so he studied techniques for making computer concepts more meaningful. Also, Spohrer and Soloway (1986) have been developing a process model of novice bug generation. Pea and Kurland (1983), McBride (1985), and Mawby (1984) have already been mentioned as examples of research on children's poor understanding of high-level programming concepts. The focus of our model has been on identifying good strategies so that the goals for instruction and assessment will be clear.

2.3.3. Debugging is not directly taught

Although debugging is a complex cognitive skill whose acquisition requires substantial cognitive capacity, it may not be difficult to learn. The important question of whether or not children can be directly taught to debug programs is completely open because *debugging is rarely an explicit part of a LOGO curriculum.* That is, although children in a typical LOGO course may get exposed to debugging in the process of getting their programs to work, they do not get explicit instruction in detecting discrepancies, inferring error sources from discrepancy descriptions, and so on.

Gugerty and Olson (1986) reported that in a debugging situation expert programmers are better able to comprehend code and generate high quality hypotheses about possible bugs than are novices. They suggest that these skills must develop as a result of experience since debugging requires tremendous knowledge. They imply that being a good programmer is a prerequisite for being a good debugger.

However, Kessler (1986) has shown that it is possible for LISP students to learn to debug simple functions before having experience writing them. It may be possible, then, to teach good debugging skills to novice programmers so that the frequent bugs they make will not be such a stumbling block to their learning process.

2.4 Applications of the model

By doing a detailed task analysis of debugging skills, we have been able to specify the component processes, heuristics, and sub-skills that programmers must learn in order to debug well. We have also established that students do not learn the central components of the model spontaneously. Similar difficulties with debugging have also been demonstrated in the adult programming literature. The usefulness of the model does not end with characterizing difficulties, however. Our performance theory can be used as the basis for designing curriculum to teach components of the debugging model as well as for designing transfer tests and measures on which acquisition of the model's skills will be demonstrated.

2.4.1. Designing instruction

The objective of debugging instruction is to train students to use the model's debugging procedure, especially the initial phases where cues to the bug's identity and the program structure are gathered to narrow the search for the bug. The model's goal structure could

be introduced as a step-by-step procedure, after a little rewording for elementary students. Similarly, students could be taught to ask themselves the same questions the model asks the user. The specific heuristics the model uses to map discrepancies onto likely bugs and to focus search on particular parts of the program could be taught to students directly. Such instruction would decompose the complex debugging task into simple steps comprehensible by novice programmers.

In addition to teaching the content of the model, instruction must take the students' memory capacity into account. Providing memory aids such as posters with commands, debugging steps, and discrepancy-bug mappings would help to reduce the memory load to a manageable amount.

2.4.2. Predicting and assessing transfer

Given that students can, in fact, learn the debugging skills we have modeled, they should be able to use them in other contexts in which they are recognizably appropriate. Students who learn debugging in the context of a LOGO graphics course would be likely to recognize LOGO list-processing as a domain where their debugging skills would be useful, and vice versa. Our model shows that the goal structure is identical for debugging graphics and list-processing programs, as are several of the discrepancy-bug mappings (wrongpart, extrapart, missingpart, howto, whatto, and novalue in our model) and the program structure cues. In addition, the sub-skills required by the debugging process are similar in these two LOGO domains.

More generally, the debugging skills students learn in the context of LOGO programming could be recognizably useful in non-programming tasks, particularly those requiring extensive search. In a non-programming domain, the sub-skills and specific discrepancy-bug mappings are not likely to be similar to those used in programming. However, the five phase goal structure would be similar to other debugging situations as long as the desired outcome, buggy directions, and buggy outcome are available to the solver. Also, if the buggy directions are structured in ways similar to LOGO programs, then the program structure cues should also be similar. The transfer tests used for the research discussed in this dissertation were designed with these criteria for similarity in mind.

The measures used to assess debugging skill were also based on the model. When using the model, greater knowledge input results in narrower search (fewer goals);

developing debugging skill should therefore result in decreased debugging time. If all the knowledge input to the model is accurate, the bug can be located and corrected all in one cycle through the model (from the initial goal to test the program to the final retest goal). Developing accuracy in debugging should therefore result in fewer debugging cycles needed to locate and correct bugs. These measures of speed and efficiency, along with some qualitative characterizations of debugging strategies, will be the core of our transfer analysis.

The details of the model-based instruction and assessment will be described in Chapter 3. The primary goal of the investigation was to assess the extent to which students trained to debug one type of LOGO programs could transfer their skills to a second type of LOGO programs and to debugging of non-computer directions.

3. Designing model-based instruction and assessment

The formal task analysis of LOGO debugging skills presented in Chapter 2 provides a basis for detailed instruction and assessment of those skills. This experiment was designed to address several issues. The first goal was to discover whether students can learn the debugging skills used by the model when those skills are taught directly. The second goal was to demonstrate that debugging skills, once learned, are transferable to tasks requiring similar skills. These issues were addressed in the context of a 50 hour LOGO graphics and list-processing course taught to 22 8- to 11-year-old children in a Montessori school over a 6 month period. This chapter will discuss the experimental design, the relevant methodological issues, the instructional techniques, and finally the assessment techniques.

3.1 A combination of transfer designs

The design of this study is a combination of two common transfer designs. A savings design was used to assess the learning of programming, debugging, and editing skills in one LOGO mini-course (graphics or list-processing) and the transfer of those skills to the other mini-course. A pre-test/post-test design was used to assess the transfer of debugging skill learned in a LOGO environment to non-computer debugging tasks.

3.1.1. A pre-test/post-test design

A typical pre-test/post-test design includes a within-subjects comparison of performance before and after some treatment and a between-subjects comparison of one or more treatment groups with a control group. (See Figure 7a.) This type of design is useful when several versions of the target test are available so that versions can be counterbalanced with test time. Researchers using this design to investigate transfer generally try to show that the treatment group's performance on the target task changed while the control group's did not; they would then attribute the differential change to transfer from the source domain. Figure 7b shows the predicted results for a case in which improvement means an increase in some performance measure, such as the number of correct responses; the graph could be turned upside down to show the effect for cases in which improvement yields a decrease in the measure (e.g., solution time). All of the LOGO transfer studies described in Chapter 1 basically have this design, though some (Dalbey and Linn, 1984; Gorman and Bourne, 1983; Degelman, 1986; Brown and Rood, 1984; Clements and Gullo, 1984; and Clements, 1985) lack a pre-test, or a control group, or both.

Insert Figure 7 about here

This study includes a within-subjects pre-test/post-test comparison of performance on a non-computer debugging task. (See Figure 7c.) A mid-test was also included to monitor transfer between mini-courses. At each of the three test times, each student took three types of tests (1, 2, and 3 in Figure 7c), all of which involved debugging a written set of instructions about how to achieve a well-specified goal. There were three versions of each type of test so that the tests could be counterbalanced with test time. In other words, one-third of the students took each version at each test time (a, b, or c in Figure 7c). The hypothesis was that students' ability to debug these non-computer tasks will improve as a result of learning debugging in LOGO.

Several types of control groups could have been used in this study; however, no control group was included for the following reasons. One alternative hypothesis is that students would improve on the post-test purely as a result of their natural development over the time span of the LOGO course. The control for the effect of maturation is built into the treatment group since the age range of the students is 3 years while the span of the study is only 6 months. In other words, if developmental change over the 6 month study caused the change, then the older students should be better on the pre-test than the younger students are on either the pre-, mid-, or post-tests. (See Figure 8a.)

Insert Figure 8 about here

Another alternative hypothesis is that the improvement on the post-test is due merely to learning how to do that type of test. This hypothesis can be tested without a control group by comparing the improvement on tests given within one session to the improvement between sessions. Figure 8b shows the contrasting effects. If the improvements can be explained simply by familiarity with the tests, then the improvements from one test to the next should be constant. However, a pure transfer effect would yield improvements only between sessions (between tests 3 and 4 and between tests 6 and 7).

Having a control group with students in an identical LOGO course without the debugging instruction would be appropriate for showing that the instruction is necessary for learning to occur and that without learning no transfer occurs. Such a group was not included in this

study because the pilot study (described in Chapter 2) already showed that students in a LOGO graphics course did not learn effective debugging skills without instruction; therefore, no transfer could be expected. Unfortunately, the transfer tests were not used in the pilot study so the tie between the lack of learning and the lack of transfer cannot be shown directly. However, some suggestive evidence from this study and several concurrent studies will be discussed in Chapter 7.

3.1.2. A savings design

A savings design is useful for testing transfer to target domains where multiple versions of the test are not available, as in many problem-solving and skill acquisition domains. In the simplest case, the design involves a between-subjects comparison of two groups. One group does task A and then task B; the other group does the two tasks in the reverse order. (See Figure 9a.) Transfer is then measured as the savings the group doing each task second experienced, compared to the group doing that task first, as a result of experience with the other task. For measures such as time or the number of errors, savings would be reflected by a decrease in the performance measure. For example, in Figure 9b, better performance of group 1 than group 2 on task B would be attributed to transfer from task A. Better performance of group 2 than group 1 on task A would be attributed to transfer from task B. For other performance measures, such as accuracy or the amount accomplished, savings actually yields an increase in the performance measure. Also, the actual relationship between the lines would have to be predicted based on the relative difficulty of the tasks and the expected amount of transfer. In some cases, there may be more asymmetry than is depicted here. This design has not been used in the LOGO literature so far but has been used frequently in other transfer studies; for instance, Smith (1986) showed savings of total time for solving Tower of Hanoi isomorphs, and Singley and Anderson (1985) reported savings of total time, number of keystrokes, residual errors, and seconds per keystroke for learning a second text-editing system.

Insert Figure 9 about here

In this study, there were two groups of subjects. Both groups received the same LOGO instruction including explicit instruction in debugging. However, one group began with graphics and then moved into list-processing, while the other group took the two mini-courses in the reverse order. (See Figure 9c.) Performance on programming, debugging,

and editing (a, b, and c in Figure 9c) was measured at three times during each mini-course (1, 2, and 3 for graphics and 4, 5, and 6 for list-processing in Figure 9c), so students took a total of 6 programming, 6 debugging, and 6 editing tests. Tests were not counterbalanced with test time since they corresponded to the concepts being learned at that period in the course. Better performance on graphics tests by students taking graphics second than by students taking graphics first can be attributed to transfer from their list-processing mini-course. Likewise, better performance on list-processing tests by students taking that second than by students taking it first can be attributed to transfer from their graphics mini-course.

Figure 10 shows the combined design. Students took one version of each type of transfer test at each of three times: before the first mini-course began, between the mini-courses, and after the second mini-course ended. Performance at the three test times will be compared to show whether the debugging skills students learn from the LOGO courses transfer to debugging non-computer directions. All of the students received the same LOGO treatment including explicit instruction in debugging; however, students took the two mini-courses in different orders. During each mini-course, learning of programming, debugging, and editing skills was monitored at three times. Performance of the two groups on the same tests could then be compared to show whether the skills transfered from one mini-course to the other and whether this transfer was equal or asymmetric.

Insert Figure 10 about here

The following sections will describe various parts of this design in more detail: subjects, instructional methods and curriculum, data collection issues, and assessment procedures and materials.

3.2 The classroom and the classes

The LOGO mini-courses were taught at the Montessori Center Academy in Glenshaw, Pennsylvania during the 1985-1986 school year. The experimenter, who was already an experienced LOGO teacher, served as the instructor. The school had 3 computers prior to the experiment (one in each of the elementary classrooms and one used by the headmistress), but there was no computer instruction in the curriculum. In fact, the computers were used only rarely, and then for programmed instruction and computer games.

All instruction took place in a dedicated computer room. The classroom had two Apple IIc computers with Apple LOGO II. Students came to computer classes in six groups of four and worked in pairs. Groups were assigned by the students' regular teacher; she was given only the specification that the groups should be as mixed as possible (boys and girls, younger and older students) so that any differences between the groups would be attributable to the treatment rather than the age or sex distribution. Three of the groups took the graphics mini-course first and then the list-processing mini-course; the other three groups did the reverse. Each group had two one-hour LOGO classes per week for 25 weeks.

All of the 3rd - 6th grade children at the Montessori Center Academy participated in the experiment.[4] Twenty-two children (8 females and 14 males) ranging in age from 8;2 to 11;8 successfully completed both LOGO mini-courses. Two of the original 24 children did not complete the course; one student left the school and one moved down to 2nd grade. They were replaced by two other children who moved up from 2nd grade; however, the data from these four students are not included in the analysis. Table 4 lists the subjects, their ages, their grades in school, their standardized achievement scores, whether they had a computer at home, and whether they took graphics or list-processing first. Group A took graphics first and then list-processing; Group B took the mini-courses in the reverse order. The averages for each group show that Group A is slightly younger than Group B but performed better on the standardized tests.

Insert Table 4 about here

3.3 Instruction techniques

Instruction consisted of 2 LOGO mini-courses, one to teach graphics and one to teach list-processing. Students received 50 hours of instruction. The first mini-course was 27 hours and the second was 23 hours; the difference was due to the absence of introductory computer familiarization and faster progress during second mini-course. In addition to lessons about the programming language, students also received lessons designed to teach the model's debugging skills directly. These lessons involved specific instruction about the model's goal structure, discrepancy-bug mappings, and location clues.

---

[4]The Montessori philosophy emphasizes multi-level classrooms; at this particular school, the 3rd-6th grades were combined into a class with one teacher.

Skills in programming, debugging, and editing were tested at three times during each mini-course. Each subject took a total of 6 series of tests. In addition, each subject was given a pre-test, a mid-test (between the two mini-courses), and a post-test to determine their ability to debug non-computer directions (such as directions to set a table or distribute wages).

3.3.1. LOGO skills

Tables 5 and 6 show the sequence of instruction and LOGO skill tests for graphics and list-processing. The exact timing of the lessons varied since the second mini-course was shorter; however, the order was consistent. Spaces in the second mini-course column indicate lessons which were skipped or were incorporated into other lessons as extra time permitted.

Insert Table 5 about here

The graphics mini-course began with an introduction to interactive use of LOGO with the basic commands to move and turn the turtle, manipulate the turtle's pen, etc. Students were then introduced to procedures for writing and editing programs. During lesson 5 (3 for the second mini-course), students learned about the REPEAT command and its usefulness for making regular shapes and curves. In the 14th hour (11th for the second group), students were introduced to local variables. During lesson 21 (16 for group B), students were taught how to write recursive procedures and use conditional stop statements.

Insert Table 6 about here

The list-processing mini-course also began with an introduction to the interactive use of the basic command, the PRINT statement. Students learned the distinction between words and lists and the punctuation necessary for each. This introductory material was followed by instruction in writing and editing programs. During lesson 3 (1 for the second group), students were introduced to the MAKE command for setting global variables and to conditional equality statements. In the 14th hour (12th in the second mini-course), students were introduced to recursion, the FIRST and BUTFIRST commands, conditional stop statements, and counters. During lesson 21 (16 for group A), students were taught about generating random numbers and choosing random items from a word or list.

All lessons were taught in a guided discovery manner and included time for self-initiated projects. The intervention of the teacher in the students' work was kept to a minimum, but new commands and ideas were introduced in a structured way and beginning activities for using them were initiated by the teacher. Since the memory load of early programming is high, reminders of all commands and concepts were posted on a large bulletin board which could be viewed easily by the students. Also, an angle wheel was provided to aid in calculating relative angles and a coordinate chart was provided to aid in position and distance estimation. To further decrease the difficulty of graphics programming, students were taught computation heuristics such as angle addition and the use of symmetry.

In addition to the LOGO content in the mini-courses, students were strongly encouraged to use subprogram structure in their programs. Subprograms were introduced in lesson 6 (4 for the second group), before the first skill test sequence. An entire lesson on the benefits of using subprograms (decomposition, reusability, and compartmentalization) was presented in the lesson immediately after the first programming test.

3.3.2. Debugging skills

The "cognitive objective" (Greeno, 1976) of the debugging curriculum was to get students to acquire the same goal structure as the model. We hoped to train students to look for and to use cues for identifying, locating, and correcting bugs so that they could avoid the frustration of serial search. With only slight rewording of the goal structure shown in Figure 3, particularly the interactive prompts the model gives the user, we were able to produce a step-by step debugging procedure to teach the students. Figure 11 shows the debugging procedure students were taught in terms of the flow diagram of the GRAPES model to highlight the similarity between the model and the instruction. Debugging skills were introduced explicitly after the first debugging test (6-8 hours into the course). This timing, directly after students had experienced the difficulty of debugging, insured their understanding of the great usefulness of the skills being taught. After the step-by step debugging procedure was introduced, the students used the debugging steps to correct the same program they had tried to debug the day before.

Insert Figure 11 about here

Beginning during this lesson and continuing throughout the course, the class accumulated

a list of discrepancy-bug mappings and useful location clues. These discrepancy-bug mappings are equivalent to the knowledge in the **propose** productions and the location clues are equivalent to the knowledge in the **represent** and **specify** productions.

During this initial lesson, and any other time the students needed help debugging, the teacher used the following 4-step sequence of approaches to prompt students developing knowledge and skills: query, coaching, reflection, and recording.

The initial query approach was designed to help students to consider available cues for identifying and locating the bug. For teaching discrepancy-bug mappings, questions focused on the difference between the output and what was desired, what command(s) generally cause that type of difference, and in cases when there was more than one possibility, how to distinguish between them. Linn and Fisher (1983) suggested a similar approach to emphasizing debugging that consisted of requiring students to propose at least two hypotheses about the identity of the bug before looking into the program. For teaching structural cue use, the questions focused on knowledge about how the program was set up and where the buggy command might be in the program (near what other command?, before/after what other command?, in a REPEAT statement?, or in a subprogram?). Coaching was used essentially as a memory aid following the query process; it merely consisted of reminding the students of the clues they had identified.

Reflection following debugging emphasized useful clues from each debugging episode. Students were prompted to recall what difference they were trying to fix, what type of bug had caused that problem, and whether similar differences were always caused by similar bugs. For structural cues, students were asked how the program was structured, where the bug had been found in that structure, and what clues did (or could have) helped them to locate the bug more easily. This prompting was designed to help students develop abstract rules about the discrepancy-bug mappings and ways to distinguish them as well as about useful structural cues. The recording approach was included to keep a written record of the discrepancy-bug rules for use in later debugging situations. The records were kept on large charts with the headings: "If this goes wrong," and "Then check for this bug." Graphics and list-processing classes kept separate charts on two sides of a mobile bulletin board. The graphics side was not visible during list-processing classes nor the list-processing side during graphics classes so that the students could not learn the mappings for a mini-course they were not currently taking.

During the entire mini-course, only one half-hour lesson was devoted explicitly to debugging. (Transcripts of one graphics and one list-processing lesson from the first and second mini-course presentations are provided in Appendix III.) During the rest of the course, posters with the debugging method, the discrepancy-bug mappings, and the location clues were available. Also, students were continually prodded to use the debugging skills and challenged to find new mappings.

3.4 Data collection issues

The primary goal of studying skill acquisition and transfer is to understand the detailed mechanisms and internal structures of cognitive processes involved. Several methods were used to ensure collection of data that would facilitate this understanding. The students' behavior on all of the tests was videotaped to get a detailed record of the intermediate steps in the solution processes. In addition, students were encouraged to think aloud so that the goals, strategies, and knowledge influencing their solutions could also be recorded. To encourage thinking aloud, students worked in pairs for some of the tests. Also, to prevent students from getting stuck on any one part of a test, the experimenter intervened to provide help when impasses were reached. Each of these methodological issues will be discussed in more detail in the following sections.

3.4.1. Protocols

In order to study cognitive processes, the data collected must include more than information about the end-product of the process. The intermediate steps in subjects' solutions reveal the path(s) by which they reached them. Each step in the process may depend on having particular knowledge about the current situation and may produce new knowledge as well. Think-aloud protocols were used to solicit verbal expression of the knowledge currently active in the subjects' short-term memory (Ericsson and Simon, 1984).

To capture these intermediate steps and relevant knowledge, all of the tests were videotaped. For all computer tests, the camera was focused on the computer screen only. The videotape contained a visual record of all screen activity and an auditory record of all verbalizations by the subjects and the experimenter. For the non-programming tests, the camera was focused on the subject's paper(s) instead of the computer screen. In both cases, the videotape also contained a record of an elapsed time indicator (accurate to the nearest second). These recording measures should provide the information needed to

accurately assess the complex processes involved in computer programming and debugging, in particular, information about the knowledge subjects had (especially that which would be considered input to the operators in our model) and the goals driving their solution process.

3.4.2. Partners

From our past experience, we expected that the children would have difficulty giving think-aloud protocols, especially in such a cognitively demanding situation as computer programming and debugging. For this reason, children worked in pairs during class and for some of the testing. We felt that the joint effort would require communication of strategies for and knowledge about the task. This collaboration was used primarily for the benefit of the experimenter. It would have been interesting to study the effects of joint work; however, the current study was not designed for that purpose. Thus, one untested assumption in this study is that the use of partners has not distorted our results.

There are several justifications for this assumption. First, Montessori instruction stresses collaboration so the students in this study were used to working in pairs and small groups. Also, pairs were chosen such that children always worked with a partner of equal ability (usually also of equal age). Research on collaboration in programming classes is scarce. Webb (1984) showed that students' mastery of BASIC concepts was equal when they worked in pairs and when they worked alone but admitted that the learning processes may have been quite different. On the other hand, Jacobson and Jackson (1986) taught business students a course in computer programming either with small amounts of peer review or with an equivalent amount of additional instruction. They found that the students who participated in the peer review process had higher scores on a content test and, more importantly, used only 60% of the computer resources that the control group used. Hawkins (1983) makes similar claims for the positive impact of collaborative work on cognitive and metacognitive skills but has not demonstrated the effect.

3.4.3. Prodding

In an attempt to minimize frustration and maximize data collection, the experimenter did provide help when students reached an impasse or had gone far afield of effective procedures. For example, if students had been allowed to focus on only one bug for the entire test time, then we would have been able to collect data only about their solution process for that one bug. Intervention was desirable so that students would have an

opportunity to attempt as many bugs as possible during the allotted time, thereby maximizing the amount of data collected.

When using such intervention, it is important to consider its effects. Therefore, the frequency and type of intervention was noted for all tests and will be discussed in the analysis section.

3.5 Procedures for assessing skill acquisition

Transfer is not possible if learning has not taken place. In order to specify precisely which skills were available for transfer to the non-programming tasks, skill development was monitored three times during each mini-course and the savings from one mini-course to the other was measured and then tested for significance using two-way ANOVA (1st mini-course vs. 2nd mini-course and graphics vs. list-processing).

Three types of tests were used to monitor three related, but distinguishable, skills. At each test time, students took three tests, each of which used the same program goal. First, they wrote a program to accomplish the specified goal. Then, they debugged a purposely buggy version of the experimenter's program to accomplish the same goal. Finally, they edited the experimenter's program. These three types of tests are represented by the a, b, and c in Figure 9c.

Each student took three series of these tests during each mini-course (1, 2, and 3 in Figure 9c). The first programming test was taken after students had some experience with subprograms but before the teacher had stressed the usefulness of subprogramming for simplifying the debugging process. The first debugging test was taken after special attention to subprograms but before debugging instruction. These first tests therefore serve as a baseline, that is, the level of skill prior to explicit instruction. The following sections characterize each type of test and describe the administration procedures.

3.5.1. Programming tests

For programming tests, the students were asked to write the program(s) to accomplish some particular goal. This type of test was included to monitor developing programming skills and to see what debugging skills were used spontaneously during programming. In addition, the programming experience provided the students with advance understanding of the program(s) that would be encountered in the debugging test. This procedure decreased

the comprehension demands of the debugging process, i.e., the children were already very familiar with what the program should do and the general way it which it should be done. Also, familiarizing the children with the program added to the content validity of the debugging tests since most debugging takes place in a context where the programmer is already familiar with the program.

Students worked in pairs for the programming tests. They were given a program plan (a pictorial description in graphics or a written and oral description in list-processing) and asked to write the program(s). The students' programming was videotaped until the program was complete or one class period had elapsed, whichever came first. The experimenter intervened only when students were at an impass or were going far afield of the desired goal.

The programs used in these tests were designed to require use of all the programming concepts learned up to that point. Graphics tests all included portions which could most efficiently be done using a repeat statement. Later tests included portions which should have used variables and recursion. Graphics students wrote programs to draw a farm (with farmhouse, silo, and four identical cornstalks), a sea scene (with boat, sun, and seagulls of various sizes), and a garden (with two rows of flowers decreasing in size and two different sizes of butterflies). List-processing students wrote programs to play madlibs with a user, to play an unscrambling game with a user (using recursive stepping through a list), and to generate poetry (with some user input and some random selection). All of the programs (graphics and list-processing alike) could have been written using multiple subprograms. In general, the programs provided opportunities for the students to demonstrate their most advanced skills. The program plans and the experimenter's correct programs for accomplishing those plans are presented in Appendix IV.

3.5.2. Debugging tests

For debugging tests, the students were asked to debug a buggy version of the experimenter's program to accomplish the same goal as used in the programming test. These tests were designed to assess the students' acquisition of the model's skills in a situation where debugging was required. These tests used programs that the students had not written themselves so that the bugs would be the same for all students.

For each test, the students knew what the program should do since they had written their

own version of it already. Working in pairs, students were given the buggy programs online and asked to fix all the bugs. The students' debugging was videotaped until the program worked or one class period had elapsed, whichever came first.

The programs used for the debugging test were well-structured; in other words, they made appropriate use of subprograms and other LOGO substructures such as repeat and recursion. Six bugs were added to each program. For the graphics tests, five of the bugs were semantic bugs while only one was a syntax bug. Syntactic errors include misspellings, inappropriate punctuation or spacing, and other errors which interrupt the running of the program. Semantic errors do not stop the program from running but do cause faulty output. Since syntax tends to be more of a problem for list-processing, those tests contained three syntax and three semantic bugs.

The bugs were chosen so that the discrepancies they caused would be fairly independent either in space (usually for graphics) or time (usually for lists). The only other criterion for bug selection was that there be a variety of discrepancy types in each program (size, orientation, location, extent, missingpart, etc. for graphics and non-matching, wrong variable, printed variable, etc. for lists). No attempt was made to control for the difficulty of the programs or the bugs since the level of programming skill was developing. Appendix V presents the buggy programs and the buggy output for all six debugging tests (three graphics and three lists).

3.5.3. Editing tests

For the editing tests, the student was asked to make the changes marked on a printout of the program from the debugging test. These tests were designed to monitor separately the developing skills for interacting with the computer and thereby to determine how much of programming and debugging time is merely due to typing and editing. They allow us to document whether improvement of debugging times could be due merely to improved typing and maneuvering skills. That is, improvements in gross performance may simply be due to the improved "clerical" skills of manuevering in the editing environment.

Students worked individually on the editing tests. The student was given the buggy programs online and a hardcopy of the buggy programs with the 6 corrections marked clearly in red ink. Each correction was pointed out to the student who was then asked to make all the changes and run the program to demonstrate that it worked properly. If the

program did not work, either because one or more of the changes had not been made or because the student had introduced new bugs, the student was asked to correct the errors and re-run the program until it worked properly. The entire editing test was videotaped.

The programs used for the editing tests were the same programs used in the debugging tests. Students were already familiar with the program and may have even remembered some of the bugs they had found on the debugging test. The marked hard-copies are presented in Appendix VI.

3.6 Procedures for assessing debugging transfer

Transfer is "simply applying information about a known category to a new instance" (Bassok and Holyoak, 1986). The goal of the transfer assessments is to discover which of the knowledge and skills available for transfer are actually applied in new instances. The new instances must, therefore, be recognizable as members of the same category and the skills must indeed be applicable.

Choosing an appropriate "far" transfer task rarely has a principled basis. Typically, researchers have a plausible but vague notion of the similarity between the source and target tasks (e.g., McGilly et al., 1984). The debugging model, described in Chapter 2, provides a way to advance the specificity of predicted transfer effects and the choice of tasks for debugging skills.

Transfer can only be successful when the new domain is viewed as an instance similar to the base domain in a way that would make use of already learned skills appropriate. The choice of transfer tasks must be based on the similarity of the required skills. For this reason, the transfer test used in this study was designed to be similar to LOGO debugging in terms of the debugging situation, the types of information available, and the location cues available. The tests were designed only to test the transfer of debugging skills (not programming or editing skills) since debugging skills were the subject of our task analysis and explicit instruction.

3.6.1. Similarity between debugging programs and debugging directions

Three types of transfer tests were designed, all of which involved detection and correction of errors in a written set of instructions about how to achieve a well-specified goal. This task is similar to LOGO debugging in three ways.

First, before the transfer tests began, the teacher gave instructions which were designed to highlight the debugging nature of the tasks. Program debugging is viewed as a situation where a programmer has given the computer commands, the computer follows the commands perfectly, but something goes wrong because one of the commands is wrong. The debugger's job is to find the bug in the commands and fix it so that next time it will run correctly. The instructions for debugging directions mimic the program debugging situation:

> "Today I would like you to read three stories. In each story, someone gives someone else directions. The person follows the directions perfectly, but something goes wrong because one of the directions is wrong. Your job is to find the problem with the directions and fix it so that next time it will be done correctly."

The cover story for each item reiterated these instructions.

Second, information about the desired and actual output is available in debugging situations even before the programs are viewed. For debugging directions, similar information was provided before the directions could be viewed. In two of the three test types, this information was in the form of pictures; however, in the third, it was in the form of tables. From the pictures and tables, subjects could have gathered clues about the identity of the bug and the probable location of the bug just as they could in the program debugging situation.

Third, the lists of directions were structured in ways similar to LOGO, primarily like subprograms but also like repeat statements. This was accomplished by the addition of headings between sections of directions to label their purpose. Headings were printed flush with the left margin, whereas the directions relating to them were indented. Subjects could use the headings to determine which sections of the directions were likely to contain the bug just as they could use the subprogram names to guide their search for program bugs.

3.6.2. Possible solution strategies

The following example will show how the model's brute force strategy (low information search) and selective search strategy (high information search) would solve one of the transfer items. Figure 12 shows the plan and outcome for the furniture arranging problem. Table 7 lists the accompanying directions. Before viewing the figure, students read the following cover story.

> Mrs. Fisher was moving into a new house with the help of two movers. She asked them to arrange the furniture in her house and gave them a list of directions to follow. The movers followed the instructions perfectly, but there was one problem with the directions so the furniture was not arranged correctly.
>
> The next page shows the way Mrs. Fisher wanted the furniture to look and the way it looked after the movers arranged it. Use these pictures to help you find the problem with Mrs. Fisher's directions. Then fix the directions so the movers could arrange the furniture correctly.

Insert Figure 12 about here

Comparison of the two floor plans reveals that there is a table out of place. Closer inspection may reveal that the table is in the living room. One might also notice that the table has been placed between two chairs in both drawings and hypothesize that the confusion resulted from a misunderstanding of which two chairs. All of the directions begin with "Here are the directions so-and-so gave to so-and-so." and end with "Change or add one thing to fix so-and-so's directions." In most cases, the directions are divided into three parts; here, one part describes how to arrange the dining room, one part the living room, and one part the kitchen.

Insert Table 7 about here

Solving this problem would be quite tedious for someone using a brute force strategy, as many children did. The solver would read each line and check the picture to make sure it was correct until the incorrect direction was located. A solver who knows to look for a misplaced table might scan the directions until reaching one describing the placement of a table. This would lead to false alarms on the lines describing the three other tables in the home (especially the two which are described prior to the correct table). A solver who knows to look in the directions for the living room will ignore the dining room directions and

focus only on the living room ones. Depending on the other available information, the solver might check each of the living room commands or only the ones referring to tables. A solver who noticed that the table was between two chairs could scan for a phrase about a table between chairs. Since the kitchen directions follow the incorrect direction and are independent of it, they would be ignored by all solvers who locate the bug on their first pass through the directions.

Solvers with all of these strategies could locate the bug and add the information to define which two chairs the coffee table should be between. The search process, not the success rate, is what should distinguish the different strategies. However, solvers who search more of the directions might be more likely to false alarm and therefore be less successful. The remaining eight tests are presented in Appendix VII.

3.6.3. Transfer tests

The three types of tests were chosen, on the basis of informal pretests, to get a range of difficulty. Three items of each type were constructed, one each for the pre-, mid-, and post-tests. An equal number of subjects were given each item at each test time. The easiest problems involved directions for arranging something (setting a table, building with blocks, or arranging furniture). The next easiest problems involved directions for distributing something (paying wages, delivering trees, or ordering food). The most difficult problems involved directions for traveling somewhere (playing golf, visiting airports, or running errands). The tasks were always presented in order of increasing difficulty so that students would not do poorly on an easier test purely as a result of being frustrated by a harder one.

The transfer tests were given in the computer classroom at the teacher's table. Students worked on the tests individually and were asked to "read and think aloud" while they worked. All work was done in a test packet which contained the three stories, pictures or tables comparing the desired outcome with the actual outcome, and a list of directions. The entire transfer test was videotaped. In this case, the camera was focused on the test packet as it lay on the teacher's table; the teacher made sure the student kept the packet flat while working.

The videotaped records of the students' performance were used to determine their search strategies in addition to the accuracy and timing of their answers. The significance of pre-, mid-, and post-test differences in qualitative strategy classifications will be tested using a χ^2

analysis, and the significance of differences in quanitative analysis will be tested using a two-way ANOVA (test time by test type).

To summarize, the experiment is a combination pre-test/post-test and savings design in order to assess the transfer of debugging skills (and support skills) from one LOGO context to another and to assess the transfer of debugging skills from the LOGO contexts to non-programming contexts. The debugging model, described in Chapter 2, was used extensively for designing the instruction and the transfer tests so that learning and transfer would be maximized. Chapter 4 describes the savings from the first mini-course to the second and Chapter 5 discusses the transfer of debugging skills to non-programming domains.

4. Skill acquisition

The savings seen on the second mini-course as a result of having taken the first reflects the amount of transferable skill the students learned. Two clear, but relatively uninteresting, savings have already been mentioned: the students did not need to relearn how to use the computer and they progressed more rapidly through the lesson sequence so the second mini-course took only 23 instead of 27 hours to complete (85% of the first mini-course time).

Students were expected to recognize the second LOGO mini-course as a new instance where previously learned programming, debugging, and editing skills could be applied. To the extent that the application is appropriate, the transfer should be positive. Transfer would be negative in a case where application of previously learned skills was detrimental in the new situation.

Many skills would be expected to transfer from the students' first mini-course to their second. Most obviously, the second mini-course took less time because the introductory computer familiarization was not necessary (learning to insert the disk, turn on the computer, run the printer, etc.) Also, graphics and list-processing use the same editor so the editing commands are identical. Procedures are run in the same manner, except that students must type CS (clearscreen) before each graphics run to reset the screen from the last one. Several programming concepts might also be expected to transfer, such as the use of subprograms, variables, and conditional statements. Some rudimentary syntactic skills would be expected to transfer: the importance of correct spelling, spaces, and perhaps the use of a colon before a variable, but beyond that, list-processing has many syntactic rules which do not occur in graphics and little transfer would be expected.

Most importantly, much of the debugging skill students were taught should be transferable to the second mini-course. Certainly, the goal structure would be identical, as would most of the location clues. The discrepancy-bug mappings would differ, except for some of the syntax errors, but the procedure for using them would be the same.

4.1 Debugging skills

The goal of the debugging analysis was to document which of the model's skills the students were able to acquire from the direct instruction provided in both LOGO mini-courses. There was no comparison group (a group that got no explicit instruction in

debugging); however, the results from this analysis can be compared to the results from the pilot study (Carver and Klahr, 1986) described in Chapter 2. This section will show that when given debugging instruction based on the performance model, students were able to acquire effective debugging skills. Without such instruction, students in the pilot study debugged poorly.

Debugging episodes were transcribed directly in terms of the model's goal structure. Transcription sheets were copies of the model's goal structure with spaces where the goal names used to be. Episodes were divided into cycles based on the test goal. A new cycle began each time the subjects ran a program or ran a series of programs without doing anything else in between; these program runs were recorded in the TEST space. Comments about whether or not the outcome of a test matched the plan were noted in the EVALUATE space. Similarly, comments about the discrepancy and the proposed bug were written in the DESCRIBE and PROPOSE spaces; comments about the knowledge of the program structure and clues for where the bug might be in that structure were recorded in the REPRESENT and SPECIFY spaces. The programs students edited in their search for the bug were noted in the FIND space. Commands they read were entered in the INTERPRET space and their assessment of the commands' correctness was entered in the CHECK space. Finally, any comments about the change that needed to be made were entered in the CHANGE space, while the actual replacement was entered in the REPLACE space. The order of comments and actions was preserved by numbering each entry. The time at the start of each cycle was also entered on the transcript. In addition, all experimenter interaction with the pair of students was recorded on the side of the sheet, labeled with an E, and numbered in sequence with the other events.

Figure 13 shows an idealized transcription for the last part of the POETRY test. The example is "idealized" because subjects had difficulty continuing to give protocols so they rarely made comments about all of the goals in any one cycle; comments from a variety of subjects have been combined to give a complete example. The example transcript begins with a test of the program POETRY. A negative evaluation is indicated by the comment, "Oh no!" The discrepancy was described as, "it used the wrong name" and the bug proposed as, "it has the variables mixed up." One student asks, "which subprogram should we try?" Both know that the program representation includes subprograms. The other specifies the buggy subprogram as "goodbye." They edit the subprogram GOODBYE and scan for the variable. :NAME1 is isolated and understood to be the wrong variable name.

"It should be hello," says one student. The students then replace :NAME1 with :HELLO and exit the editor to retest the program. A trace of the model using the same knowledge to debug the same program is one of the example traces provided in Appendix II.

Insert Figure 13 about here

From the debugging transcripts, several related aspects of debugging skill were coded to probe the students' acquisition of different parts of the model's skills: achievement, speed, efficiency, clue gathering, and search strategies.

4.1.1. Achievement

One indication of debugging skill is the number of bugs students were able to fix during the allotted hour. Most of the results of the skill analyses are presented in a fashion similar to Figure 14 so it will be described in detail. Figure 14a shows the performance of groups A and B on the three graphics tests. Group A took the graphics mini-course first and group B took it second, after having had list-processing experience. Figure 14b shows the performance of groups A and B on the three list-processing tests. Here, group B took the mini-course first and group A second. The irregular pattern from test 1 to 2 to 3 on these graphs reflects the differences in problem difficulty. The tests were not counterbalanced since the programming concepts used on the tests reflected the order of instruction in the course.

There are several more important patterns to extract from these figures. The first pattern to consider is the relationship between the graphics and list-processing graphs. The second pattern to extract from these graphs is the relationship between the lines representing the two mini-courses. A consistent pattern between the first and second mini-course lines on each graph would indicate the savings resulting from the first mini-course experience. Figure 14c summarizes the results; it highlights the two important patterns (graphics versus list-processing and first versus second mini-course) and deemphasizes the irregular pattern of the three tests within a mini-course. For the purpose of this graph, the within mini-course tests are considered essentially as three trials so the scores have been averaged to get a score for each mini-course as a whole. For the remaining figures of this type, the discussion will focus only on these patterns of results.

Figures 14a and b show that during the second mini-courses, all of the students fixed all

of the six bugs within the hour, whereas, in the first mini-courses, they did not, $F(1,76) = 23.15$, $p < .01$. Figure 14c shows the aggregate results; the improvement from the first mini-course to the second mini-course was the same for the graphics and list-processing tests, $F(2,76) = .06$.

Insert Figure 14 about here

4.1.2. Speed

The debugging speed was also measured. In cases when the students corrected all six bugs, this included all debugging time up to but not including the final run (when the program worked correctly). When the students did not correct all six bugs, the time was measured up to but not including the program run that confirmed their last correct fix. This adjustment excludes time at the end of the session spent on bugs that were never corrected; thus the speed measure includes only time spent on bugs that were fixed. For all pairs, the total time was divided by the number of bugs fixed. The model makes no predictions about the absolute debugging time, but the time would be expected to decrease as debugging skill improves since strategies would shift from brute force to more focused search which requires fewer subgoals.

Figures 15a and b compare the performance of students on the graphics and list-processing tests in the first and second mini-courses. The graphics groups took almost two minutes longer per bug than the list-processing groups, $F(1,76) = 7.45$, $p < .01$. Figure 15c shows the same results in terms of the time savings on the second mini-course tests as a result of having taken the first mini-course. Savings for the list-processing tests was about half and for the graphics tests was about one-third, $F(1,76) = 42.69$, $p < .01$.

Insert Figure 15 about here

4.1.3. Efficiency

As debugging skill improves, students should also take fewer cycles (each isolated test goal initiates a cycle) to fix each bug. Perfect debugging (when clues are available and accurate) requires only one cycle to locate and correct each bug. To assess this improvement, the number of cycles per bug was measured. As with time, the total number of cycles was used in cases where students corrected all six bugs; however, the number of

cycles up to the last correct fix was used in cases where all the bugs were not corrected. Once again, the number of cycles was divided by the number of bugs fixed.

Figures 16a and b show the improvements in efficiency from the first mini-course to the second for each test, $F(1,76) = 12.64$, $p < .01$. The graphics groups took more cycles than the list-processing groups to fix each bug, $F(1,76) = 62.46$, $p < .01$. In the second mini-course, the list-processing group was averaging close to the perfect one cycle per bug. Several groups of students actually took fewer than one cycle per bug because they fixed several bugs in one program without exiting to retest the program in between. Figure 16c demonstrates the savings for each course as a whole. For both groups, the savings was approximately 1 cycle per bug.

Insert Figure 16 about here

In order to account for the differences between the two groups and for the savings from the first mini-course to the second, the debugging process was considered in more detail. The following descriptions will concentrate more on the savings between the two mini-courses rather than on differences between tests within a mini-course.

4.1.4. Pre-search clue gathering

Comments about the discrepancy, the bug, and the bug's location were scored as correct or incorrect and as made prior to or after the first command identified as the bug. As students' knowledge of discrepancy-bug mappings and of location cues increased, the proportion of correct comments should increase. The total number would actually decrease as they they improve since debugging will take fewer cycles. Also, the proportion of comments made before suggesting a command as the bug should increase if the students learn the goal structure of the model which stresses the value of seeking cues to narrow the search.

Students made very few comments overall, perhaps because the task was so cognitively demanding. Students described the discrepancy aloud for about half of the bugs, but they proposed bugs for only about 1/4 or 1/3 of the bugs prior to beginning their search. Location descriptions were more frequent than bug proposals but were still offered for less than half of the bugs. Students in the second mini-course made fewer comments about the discrepancy, the bug, and its location because they took fewer cycles to isolate each bug.

Figure 17 shows the percentage of these comments that were accurate for a) discrepancy, b) bug, and c) location comments. Accuracy was slightly higher for list-processing debugging. Comments about the identity of the bug were least likely to be accurate. The savings between mini-courses was minimal.

Insert Figure 17 about here

However, Figure 18 shows that the savings between the first and second mini-courses is in the percentage of comments made prior to beginning to search for the bug. Particularly for bug proposals, the students are learning the value of proposing the bug before searching so that the search can be selective.

Insert Figure 18 about here

4.1.5. Faulty search

At the same time, evidence of poor debugging behavior would be expected to decrease. Such behavior can be measured as the number of correct subprograms the students erroneously edit, the number of false alarms they make (correct commands identified as the bug), the number of times they abandon their search for a particular bug, and the amount of help they need from the experimenter.

The number of times the students looked into a subprogram that did not contain the bug (or information relevant to the bug such as variable values) should decrease as the students learn to use clues for locating the bug. The subprogram structure of the buggy programs was easy for the students to recognize because the subprograms that had been loaded were displayed on the computer screen at the beginning of the test. Students rarely misjudged which subprogram did which part of the program because the subprogram names were related to their function. The mean number of times subjects looked into a program that did not contain the bug ranged from 2 to 3 per test (i.e., per 6 bugs). Most of these errors resulted from forgetting the names of the programs or forgetting what subprograms existed. The amount of brute force search (reading and checking each command in a program) did decrease from the first mini-course to the second. The number of instances of brute force decreased from 28 to 14 for the graphics tests and from 13 to 1 for the list-processing tests.

Many students actually focused too much on the subprograms, forgetting that they were called by another program. For example, most students added a SETPOS command to the WAVES subprogram when the waves were positioned incorrectly rather than changing the faulty SETPOS command in the main SEASHORE program; they never looked into the SEASHORE program to see whether a positioning attempt had been made. Also, when testing subprograms independently on graphics tests, students were bothered when a figure ran off the screen even if it had not done so when the main program ran; they often added positioning commands to the subprogram even when there was no positioning bug.

The number of correct commands which were mis-identified as the bug (false alarms) were tallied. The false alarms were also categorized according to whether the students actually changed the command or only proposed the change and whether the command was in the same subprogram as the actual bug. Improved search strategies should decrease the number of false alarms, particularly those in subprograms other than the one actually containing the bug. In fact, the number of false alarms dropped from 206 to 150 from the first graphics mini-course to the second and even more dramatically from 87 to 27 for the list-processing tests. The difference between the magnitude of false alarms for the two different LOGO domains is not surprising in view of the vast difference in the total number of commands in the programs and the greater similarity between instances of each command (e.g., there are many indistinguishable FD commands in one graphics program). The percentage of false alarms in the wrong subprogram remained at 15% for both graphics mini-courses but decreased from 31% to 7% from the first to the second list-processing mini-course.

The high number of false alarms accounts for many of the extra cycles the students took to correct the bugs. Because punctuation is so complicated in list-processing, poorer students adopted a trial and error replacement strategy for fixing punctuation (brackets to quotes, quotes to brackets etc.) when they could not identify the bug.

The number of times students abandon their search (which the model would never do) should also decrease. In fact, subjects rarely abandoned their search (though this is partly a result of the availability of help from the experimenter). There were 13 instances of abandoned search during the first graphics mini-course and 10 during the first list-processing mini-course. In the second mini-course, the number of abandoned searches decreased to 9 and 4. In addition, students were more likely to restart an abandoned search in the second mini-course.

4.1.6. Incorrect changes

For each bug that was correctly identified, the number of changes it took to get the correct command was tallied. Incorrect fixes were scored as either syntactic or semantic errors. Incorrect fixes should decrease as the students become better programmers, though not necessarily as they become better debuggers.

Attempts to determine the appropriate argument for a command once the bug had been identified accounted for an average of 1 extra cycle per bug. There was no improvement in the subjects' ability to make changes once the bug had been identified on graphics tests. The debugging instruction did not provide any help for generating the correct command once the bug had been identified. For list-processing, however, the identification of the bug almost specifies the appropriate change. For example, discovering that a variable has been printed inside a list implies that the appropriate fix is to remove it from the list, whereas knowing that a particular left turn was not enough does not imply what the turn should have been, only that it should be more. Extra cycles to determine the correct fix decreased from an average of .4 per bug to .1 per bug as students became more familiar with the appropriate changes for each bug.

4.1.7. Experimenter help

Each instance of experimenter help was scored for the type of information it provided the students: description of the discrepancy, interpretation of a command, recognition of a false alarm, identification of the bug, location of the bug, specification of the change, recommendation of strategies, or reminder of something they had previously done or said. As students begin to debug like the model, they should need less help from the experimenter. In particular, as debugging strategies improve, students should need less help gathering clues to narrow their search, less help correcting false alarms (since they should be making fewer), less help choosing strategies, etc. For the list-processing tests, the amount of help subjects needed decreased markedly from 177 instances in the first mini-course to only 80 in the second. The biggest difference for the list-processing group was on the second test which included FIRST-BUTFIRST recursion, conditional statements, and counters. Apparently, the group that had graphics background had less difficulty dealing with these concepts than the group with no prior experience, perhaps because they had encountered tail recursion and conditional statements before. However, the amount of help needed on the graphics tests did not decrease: there were 150 instances of help in the first

mini-course and 164 in the second. Much of this help is related to other poor debugging strategies such as the high number of false alarms and incorrect fixes.

4.1.8. Additional testing strategies

Finally, there are many additional strategies which could facilitate debugging, such as doing extra test runs to better describe the discrepancy and specify the location of the bug, testing subprograms rather than the main program when appropriate, using mock input during test runs rather than taking the time to use correct input, making multiple changes between test runs when bugs are relatively independent, seeking additional information from the program titles or from within subprograms not containing the bug (such as looking for a variable setting), or seeking information from the mappings chart. In fact, students used these strategies only in a few select instances when they were recommended by the experimenter. Many of the students did, however, begin testing subprograms separately after being shown how useful a technique it was. Since these strategies were not part of the explicit instruction (because they are not a part of the model), it is not surprising that students did not use them well.

4.1.9. Reaction to debugging instruction

Students were eager to learn better debugging strategies when they came into class the period following the first debugging test. Debugging the first test program was difficult for students in both the graphics and list-processing groups. Many of the students never found all six bugs within the hour test period. The entire debugging lesson, including debugging the first test program using the new strategies, took only half an hour. Most of the students made comments like, "Why didn't you tell us this before?" They had experienced the frustration of brute force debugging and were receptive to more successful methods.

After the one (and only) lesson focusing on debugging, students used the new strategies frequently, especially asking themselves which subprogram was likely to contain a particular bug. They did not, however, make use of the list of discrepancy-bug mappings as frequently. Some of the more common mappings were memorized early, but many students used the reference chart only as a last resort. Nonetheless, their debugging skills were impressive by the end of the course. (They did at least as well if not better than the LOGO teachers tested by Jenkins (1986) on the same programs.)

4.2 Programming skills

The goal of the programming skill analysis was to describe the level of programming ability students acquire, particularly skills related to debugging skill. The analysis will emphasize the amount of structure students build into their programs since proper structuring makes debugging easier (Korson and Vaishnavi, 1986). Debugging strategies will also be emphasized so that the spontaneous strategies used on the programming tests can be compared with the strategies used on the debugging tests.

The data for the programming analysis included the students' programs, the output they produced, and the videotapes of their solution process. From the videotape, four classes of programming behavior were transcribed: code writing, code changing (without having tested the program), code testing, and code debugging. In addition, the students' comments about how they were structuring their program(s) and any comments made by the instructor were transcribed. Time was not measured because students rarely completed the programs within the one hour limit, especially in the graphics mini-course, and because so much experimenter intervention was required that the time measure would be difficult to interpret.

4.2.1. Achievement

One measure of the students' programming ability is how much of the goal students accomplished during the allotted hour. Each program goal was divided into units according to the following principles. For graphics programs, each major part of the picture counted as one unit. For example, the barn in the farm program had two units: the base and the roof. In addition, independent grouping of units (such as putting four corn stalks together) and positioning of units (such as placing the sun in the upper right hand corner of the screen) counted as units themselves. For list-processing programs, each query (asking a question and taking the user's input), conditional statement, response, and global variable setting counted as a unit. In addition, several types of user friendly units were scored, such as greeting the user, providing instructions, thanking the user, including WAIT statements, printing blank lines, and addressing the user by name. Driver programs counted as units for both graphics and list-processing.

Each of the programs written by the subjects was then examined to see which of the units had been completed, which had been attempted, and which had not been attempted. The achievement measure for each pair was the percentage of units they completed. As

programming skill improves, subjects should be able to complete a higher percentage of units. This section will provide the qualitative results along with a general characterization of their meaning. Appendix VIII contains the detailed description, including the total number of units for each of the experimenter's programs along with the number of student programs containing that unit.

Figures 19a and b show that students accomplished significantly more of the program goal on the list-processing tests than on the graphics tests, $F(1,68) = 58.49$, $p < .01$. However, students in the second mini-course accomplished only slightly more than students in the first mini-course, $F(1,68) = 8.32$, $p < .01$.

Insert Figure 19 about here

Graphics students began with straight-line figures and/or avoided figures with curves. This trend is evident since students tended to attempt and avoid the same units. On the FARM test, most students attempted the barn and silo and avoided the corn stalks. On the SEASHORE test, most students tried the boat and sail, and few tried the seagulls. Most students started with the flower on the GARDEN test and never had time for the butterfly.

For the list-processing tests, the students usually began with the fundamental units of the program and added the user friendly units later (if time permitted). For the MADLIB test, students began with asking the user for the necessary words and printing the story; for the SCRAMBLES test, students began with setting the global variables for the problems and then wrote the recursive function to ask the questions; and for the POETRY test, students started by asking the user for the necessary words, setting the global variables for the rhymes, and printing the poem. Many pairs wrote the base part of the program during the hour, but few students got far enough to add the user friendly units.

4.2.2. Structured programming

Another measure of programming ability is the amount of structured programming. For LOGO programs, structured programming can be defined as appropriate use of line breaks, subprograms, repeat statements, variables, and recursive calls. Using line breaks effectively is the most rudimentary structure a LOGO program can have. LOGO can handle lines of up to 256 characters so most programs could be typed without ever hitting <return>. Alternatively, commands need not be kept together so programs could be entered with a

<return> after each independent command (i.e., those that are not arguments to other commands). Neither of these strategies groups commands according to what part of the program plan they accomplish; such a strategy would be useful for later debugging and modifying. For example, many graphics programmers keep on the same line all commands executed after a PenUp command but before the associated PenDown command; this allows easy distinction between lines of commands that draw something and lines of commands that only move the turtle. Using subprograms, repeat statements, and recursion also serve to organize programs by grouping commands. In addition, using variables groups like arguments as well as allowing for similar functions to be accomplished by one program with different inputs.

Once again, the experimenter's program was used as a standard. One point was given for the use of appropriate line breaks, one point for every subprogram call, one point for every unique repeat statement (multiple uses of the same repeat statement were not scored because they should have been embedded in a repeat statement themselves or written as a separate subprogram), one point for each variable used, and one point for each recursive call. Each of the subjects' solutions was scored for structure in the same way. For comparison, the mean scores are reported in Figure 20 as percentages of the total for the experimenter's program. The standard scores are presented in Appendix VIII.

Insert Figure 20 about here

Students used more program structure in the list-processing course than in the graphics course ($F\ [1,68] = 92.04$, $p < .01$), perhaps because some of the programs required structure in order to function properly (e.g., recursing through a list requires setting the list in a separate subprogram). On the other hand, graphics figures can almost always be drawn in line-by-line fashion (however tedious this may be). For example, one student in the first graphics mini-course created one corn stalk by combining FDs and turns. She did use a repeat to make a curve, but she retyped the repeat statement six times (twice for each of three leaves) just to make one corn stalk. Then she systematically retyped all of the code for the one corn stalk to make a second one. She would have continued in this fashion if the instructor had not suggested that she move on to the silo and barn. In cases where students did use subprograms, they rarely wrote driver programs to combine them. When such programs were written, they used chain calls (A calls B, B calls C, and C calls D rather than A calling B, C, and D). Students did not improve from the first mini-course to

the second, F (1,68) = 2.49. Even students in group B who had used much program structure in the list-processing mini-course used very little after switching to graphics.

4.2.3. Common errors and misconceptions

The students misconceptions and common bugs were also catalogued in order to characterize developing programming skill. Errors and misconceptions were consistent across mini-courses; in some cases, students in the second mini-course actually made more errors or demonstrated more misconceptions because they accomplished slightly more and used slightly more advanced structuring.

The most frequent errors in graphics programming were inaccurate arguments (moving, turning, or repeating too much or too little). Students also made direction errors by turning left instead of right or vice versa. Occasionally, they forgot to type part of a command, typed part of a command twice, put spaces where they did not belong, or omitted spaces where they did belong.

The most common misconceptions in graphics related to moves between figures and drawing figures with curves. Students frequently forgot to make a move between figures, especially when using repeat to get several identical figures (such as four corn stalks) so the figures were drawn on top of each other. When they did remember the move, they often forgot either to pick up the pen, to put down the pen, or both. PU and PD commands were also frequently misplaced (after the move, for instance). Students also had difficulty remembering how to use a repeat statement to draw a curve. They frequently confused the two rules for curves: the FD distance times the repeat number equals the size of the curve while the turn times the repeat number equals the angle of the curve. Also, students had difficulty combining curves; they usually failed to make the appropriate turn between curve segments.

The most common errors on list-processing tests were punctuation and spacing errors. The most frequent of these was forgetting the parentheses around a print statement when two or more elements were to be printed. Students had difficulty understanding conditional statements, combining and separating functions (such as FIRST and BUTFIRST), and the use of MAKE for setting variables to values other than the keyboard input. They also had misconceptions about the use of variables; they frequently used the same variable name more than once (thereby erasing all but the last value), printed a variable inside a list (such

that it does not get evaluated), and confused different variable names. Students in both graphics and list-processing mis-ordered commands and made inappropriate subprogram calls (including unintended recursive calls).

4.2.4. Debugging during programming

The frequency of test runs was calculated to monitor how much the students actually make use of debugging in their own programming. Students tested graphics programs frequently, about 6 times for every program unit they accomplished. Because of the frequent testing, bugs were almost always in the most recently written commands so the debugging search was minimal. Frequently, the did not try to choose the correct command (especially the correct argument) on their first pass; instead, they used debugging as a means of determining the appropriate command. In contrast, list-processing groups tested their programs less than once per unit accomplished because they spent more time composing the phrasing of their PRINT statements. Because of this infrequent testing, they often made the same error many times and had to correct multiple bugs after each test.

4.2.5. Independence

Students in both mini-courses needed considerable help decomposing the program goals into manageable units, particularly on the list-processing tests where they had to decide what needed to happen in what order. The experimenter intervened to help with decomposition 2 to 3 times for most pairs of students. Decomposition was less of a problem for students on graphics tests, especially because they made few attempts to structure their programs. However, graphics students had particular difficulty deciding how to use a repeat statement to draw the sun and deciding when to use variables. List-processing students had the most difficulty with recursive stepping through a list (used on the SCRAMBLES test) and with deciding on the alternate paths in a conditional statement. Much of the structure used in the list-processing programs was created with the experimenter's help.

The students also needed considerable help with two aspects of debugging. They needed the most help correcting errors that had resulted from their misconceptions, in other words, in cases where their knowledge as well as their program needed debugging. Students also needed help locating bugs in cases where they had used little program structure and did not test the program frequently enough to know which part of the code was buggy. In

these cases, they may have known what was wrong, but could not find any clues for the bug's location. The next section will describe the debugging skills students demonstrated in situations where the programs were well-structured but had been written by someone else.

The quantitative and qualitative description of the students' programming abilities emphasizes that these students are still only novice programmers, even at the end of 50 hours of experience. Even though they were able to use program structure to guide their debugging of the experimenter's programs, they did not incorporate such structure into their own programs. Thus, they had more difficulty debugging their own programs than someone else's.

Katz (1986) studied debugging on one's own versus another person's programs in LISP and found nearly opposite results. He found that subjects needed twice as many hints when debugging someone else's program as when debugging their own and that they were more likely to use a backwards strategy (run the program to see what's wrong, then check each command from the last backwards) than a forwards strategy (check each command in order of execution) when debugging someone else's program than their own. The advantage for debugging one's own program may be less pronounced for the child novices in this study because they were not writing well-structured programs, whereas the experimenter's programs were well-structured. Korson and Vaishnavi (1986) also presented evidence for the greater ease of debugging structured programs in BASIC. If the subjects had been taught techniques for good program structuring, their debugging skills might have been as good when debugging their own programs as when debugging the experimenter's.

Not surprisingly, the bugs which subjects failed to find in the first mini-course were usually related to misconceptions that appeared in the programming tests. On the graphics tests, 2/3 of the subjects failed to find a wrong subprogram call (in FARM), half failed to find the orientation bug between the two curves in the seagull, and 1/3 failed to find the extent bug in the curved top of the silo. Several other bugs were missed by only one pair of subjects, usually because they ran out of time.

On the list-processing tests, the bugs subjects failed to find were again related to misconceptions they had demonstrated on the programming tests. There were three instances where a wrong variable name had been used or a variable name had been used twice; in all three cases, about half of the pairs failed to correct the bug. A little less than

half the pairs failed to find bugs dealing with conditional clauses, combining LAST and BUTLAST, printing variables inside lists, and forgetting the parentheses around print statements. Once again, several other bugs were missed by only one pair.

4.3 Editing skills

The goal of the editing analysis was to document the improvements in students editing skills so that they can be compared to the improvements in debugging time. This comparison will provide a better estimate of the actual improvement in debugging time excluding the time it takes to maneuver within the LOGO environment.

The videotapes for the editing tests were transcribed keystroke by keystroke and the total time was recorded from the lap counter on the videotape. Experimenter intervention was also transcribed.

4.3.1. Speed

Editing time was calculated as the total time up to the beginning of the correct test run divided by the number of keystrokes. The time/keystroke should decrease as the students become more familiar with the keyboard.

Figures 21a and b show that students were slower at editing list-processing programs than graphics programs ($F\ [1,128] = 8.26$, $p < .01$), though students improved in both courses. $F\ (1,128) = 7.38$, $p < .01$. The improvement represents about a 15% decrease in editing time. Despite the improvements, even the best students are only typing at a speed of 60 - 80 keystrokes (not words) per minute.

Insert Figure 21 about here

These increases in speed cannot account for the whole increase in debugging speed. The debugging time per bug was 2 or 3 minutes lower on each of the graphics tests and about 4 minutes on each of the list-processing tests in the second mini-course. This improvement is roughly a 30% savings for graphics students who had previously had list-processing and a 50% savings for list-processing students who had previously had graphics. Figure 22 shows the debugging time per bug minus the editing time per bug for each test. There is still a marked improvement in debugging speed between the first and second mini-courses.

Insert Figure 22 about here

4.3.2. Efficiency

Efficiency of editing was measured in terms of the total number of keystrokes made relative to the minimum number required. The minimum number was calculated using the following guidelines. Degraded forms of commands which were not taught yet are accepted by the LOGO interpreter were not considered. These commands actually take fewer keystrokes and may be discovered by the students. For example, LOGO will interpret ED just as it interprets EDIT; however, since LOGO is not always so forgiving (especially about spaces and paired brackets), the instruction stressed proper form. Also, short-cuts, such as those to get to the beginning or end of a line directly rather than repeating the arrow keys, were only considered when it was obvious that using them would be quicker, i.e., when the bug was at the extreme end of the line.

Students in the list-processing course were less efficient editors as well as being slower typists, $F(1,128) = 8.04$, $p < .01$. They averaged a little more than twice as many keystrokes as necessary while the graphics group averaged only 1.6 times the minimum number of keystrokes. (See Figure 23.) By the second mini-course, all of the students had improved ($F(1,128) = 39.96$, $p < .01$), reaching the same level of editing efficiency close to the perfect score of 1. Thus, the students who began with list-processing improved more than the students who began with graphics, $F(1,128) = 4.79$, $p < .05$.

Insert Figure 23 about here

Each divergence from the minimum path was categorized as one of eight types of inefficiencies: overshooting (and backing up), mistyping (and correcting), deleting too much (and retyping), placing the cursor incorrectly (and correcting), including extra spaces (and deleting them where necessary), forgetting part or all of a change (and going back to complete it), confusing editing commands or principles, and taking extra steps, which usually means either not using a short-cut when it would save a lot of keystrokes or moving around in the editor after the changes had been made. Errors were categorized in this way in order to discover whether certain errors dropped out while others persisted or whether improvement of all errors was uniform.

The students actually make more editing errors on the second and third tests than on the first test, perhaps because they are particularly careful on the initial test when they were just learning how to edit. Total keystroke efficiency does not rise with the number of inefficiencies because students may realize their error sooner and so avoid large errors. For example, as students become bolder at using the repeating key to move across the screen, they may actually overshoot the target more frequently, but may not overshoot it by as much as before since they are more aware of repeating nature of the keys. Students just learning to edit often do not realize they are making the key repeat so they overshoot by a lot before they even realize it. The students' most common errors were overshooting, mistyping, deleting too much, and taking extra steps. List-processing students also confused many commands on the first two tests. Students rarely added extra spaces, misplaced the cursor, or forgot to make part or all of a change. As students editing efficiency improved, the error rates for all types of errors dropped.

4.3.3. Incorrect changes

The number of incorrect edits was tallied as a measure of how careful the students are to check the corrections they make. (See Figure 24.) Neither group made many incorrect edits, which are errors in changes the students do not catch before leaving the editor or edits they forget to make. In the first mini-course students only made about 1 incorrect edit per 6 changes; in the second mini-course this dropped almost to zero, $F(1,128) = 7.14$, $p < .01$. Even in the first minicourse, students are careful editors though they are not particularly efficient or quick.

Insert Figure 24 about here

4.3.4. Experimenter help

On the first editing test, subjects in both groups needed the experimenter to help them (reminding them of commands, for example), but the amount of help needed decreased for both groups in the second mini-course, $F(1,128) = 35.46$, $p < .01$. (See Figure 25.) The list-processing students needed more help than the graphics students in the first mini-course ($F[1,128] = 5.55$, $p < .05$), but students in both groups needed little help from the experimenter by the second mini-course. Thus, the savings was larger for the list-processing tests, $F(1,128) = 4.01$, $p < .05$.

Insert Figure 25 about here

4.3.5. Graphics versus list-processing

The consistency of the difference between the students learning editing in a graphics environment and those learning editing in a list-processing environment was surprising, especially since they were using identical commands. Also, the groups' performance on the debugging and programming tests does not lead to the conclusion that the groups of students differ. It may be the case that editing is more difficult in a list-processing domain because of the added syntax difficulty which may cause a greater processing load. Also, both the commands and the arguments are longer so there may be more chances of misreading or mistyping even though the minimum number of keystrokes is equivalent. Another possibility is that list-processing blurs the distinction between the output [lines of text] and the program [lines of text]. The consistent finding that both groups of students reached the same level of editing skill lends support to the suggestion that learning editing in the context of list-processing is difficult. It is not more difficult to edit list-processing programs than graphics programs once the editing skills have been learned (as demonstrated by the good performance of the graphics group in the list-processing course), but learning the skills is easier in a graphics environment.

4.4 Acquisition Summary

In summary, the savings measures used in each of the skill acquisition analyses provide a context in which the level of skill acquired can be assessed. The large savings on the debugging measures indicates that students did begin to acquire the model's goal structure.

The importance of the relative difficulty of the tasks is clear from the consistent pattern of performance across various measures of the same skill, and across measures of different skills. The most prominent instance of this effect is the large decrement in performance on the second list-processing test. This test required the use of FIRST-BUTFIRST recursion, a conditional stop statement, and a counter, all of which were relatively new concepts.

The current study was not designed to evaluate the relative difficulty of various tests but rather to show the savings for students taking a test after having had previous LOGO experience and debugging experience. It is clear from the improvement between the first

mini-course and second mini-course in both graphics and lists that learning did take place. However, it is not possible to trace the within mini-course improvement. But, it is clear that the debugging skills students have acquired are abstract enough to transfer to very similar program debugging tasks. Students in the second mini-course debugged more quickly and took fewer cycles than students taking the same tests in their first mini-course. These savings were attributed to increasingly focused search. The next chapter will discuss how well these search skills transfered to non-computer debugging.

5. Debugging skill transfer

The goal of the transfer analysis was to show that the focused search strategies learned from the explicit debugging instruction in the LOGO environment would transfer to similar debugging situations not involving programming.

A more detailed trace of the solution process was available from the transfer videotapes than from the debugging tapes because on the transfer tests the students were asked to read and think aloud. For each problem, the subject's discrepancy description, bug proposal, and bug location comments were recorded as well as the type of scanning strategy they used to locate the discrepancy initially. Each line the subject read and each time the subject flipped back to look at the plan and outcome was recorded so that the search process could be quantified and the search strategy characterized.

5.1 Pre-search

The subject's pre-search strategy was one focus of the analysis. The number and proportion of correct comments about the discrepancy, the bug, the location, and possible strategies were counted. Students who learn the importance of seeking cues to narrow their search before looking at the commands in the LOGO context may transfer this practice to the new task. As with the computer debugging, students made few pre-search comments. However, the number of comments students made about the possible location of the bug in the directions increased from 9 on the pre-tests to 16 on the mid-tests to 22 on the post-tests (out of a possible 66). Also, the students needed less help describing the discrepancy between the desired and actual outcomes; instances of help decreased from 19 to 9 to 4 for the three test-times (also out of 66). Apparently, the students are paying more attention to the outcome information prior to beginning their search.

5.2 Qualitative search analysis

Each subject's reading and simulating strategies were also categorized separately as one of four qualitatively different approaches. Simulation refers to actually interpreting the direction to determine what effect it would have; usually this process involves referring to the diagrams or tables. Students were reading and thinking aloud so their search process was easily traceable. The worst strategy is to read or simulate haphazardly (a few lines here, a line or two there) or to simulate nothing (subjects must at least read something in order to

make a correction). This strategy is unlikely to successfully find the bug because it may or may not even read the buggy line let alone bother to check it against the desired outcome. A slightly better strategy, brute force, is to read every line and simulate every direction (some lines are headers not directions). This strategy is likely to find the bug but processes many unnecessary lines. Better yet is a self-terminating brute force strategy which reads and simulates everything until the bug is located and then disregards the rest. The best strategy is selective focus on the appropriate subsection of the program only, maybe even only on the part near the bug. Each subject's reading and simulating strategies for the first pass through the directions was categorized for each of the three items on the pre-, mid-, and post- tests. Subjects may have several strategies or combinations of strategies at their disposal; however, their choice of strategies was expected to shift towards more efficient strategies if they are able to learn the program debugging strategies and transfer them to the new domain.

The students' behavior on the pretests was very much like the brute force strategy of the debugging model. The predominant strategy was to read all the commands and simulate none, then to go through the directions again, simulating most of them until the incorrect direction was located. A change in focus on the later tests is apparent from the strategy classifications. Figure 26 shows the reading strategies for the students on each type of test. There were no differences between groups A and B so the data have been collapsed across group. On the pre-test, half the students read all the lines on their first reading of the directions. On the mid-test and the post-test, only a quarter of the subjects read all the lines on their first pass. A little more than a third of the students read all the directions or most of the directions until the bug was found and then disregarded the rest. A quarter of the students focused only on the directions in the same subpart as the buggy direction. Even by the mid-test, most students had shifted to a more focused strategy for reading directions, χ^2 (6) = 23.33, $p < .001$.

Insert Figure 26 about here

The simulation strategy did not shift as quickly. Figure 27 shows the strategy shift for each type of test. Two thirds of the students simulated none of the lines on their first reading of the directions on the pre-test. Thus, their reading may familiarize them with the directions, but they have not checked the directions. It could be argued that the students can remember the discrepancies between the plan and the outcome descriptions; this may

be true for the construction tests since they have simple diagrams, but it is unlikely to be true for the tables and is definitely not true for the maps which are too complicated to remember. Some students had shifted to a more focused strategy by the mid-test, but half of the students were still not simulating any commands. By the post-test, more than two thirds of the students were using focused strategies, χ^2 (6) = 33.14, $p < .001$. The shifting pattern for different tests shows that students on the mid- and post-tests were least likely to simulate commands on the construction tests where they may be able to remember the drawings. This pattern indicates that the students developed a range of strategies and know the conditions under which each is appropriate.

Insert Figure 27 about here

5.3 Accuracy

In addition, the changes subjects made were scored as either correct, incorrect, or re-writes. Re-writes were cases in which the subject did not isolate a bug but rather added directions to undo the problem and achieve the desired outcome. In the example described above, a re-write would be adding a direction at the bottom of the page saying something like, "Move the coffee table to between the other two chairs." The location of the change was scored as either correct, nearby, reasonable (on a line with an understandable false alarm, such as on another line describing a table in the example above), or incorrect. Transfer of debugging skills from computer programming would be reflected more in the improved location of the correction than in the accuracy of the correction itself since there was not instruction relevant to the particular types of corrections.

In addition to improved strategies, the students made more correct fixes on later tests regardless of the order in which they took the mini-courses. Figure 28 shows the increasing percentage of subjects who made the correct change in the correct location.

Insert Figure 28 about here

5.4 Solution time

Also, the total time from when the subject finished reading a story to the time s/he started making a change was measured. Improved efficiency of search should decrease the solution time. Figure 29 shows that the time required to suggest a bug decreases by almost half from the pre-test to the post-test as a result of the shift to the selective search strategy, $F(2,192) = 16.05$, $p < .01$. Both the accuracy and the time figures show that the traveling directions were the most difficult to debug, $F(2,189) = 19.66$, $p < .01$. They were the tasks on which the figure provided the least information about the nature of the bug and on which the directions were the most difficult to simulate. The improvements on these tasks are primarily a result of increasing use of location clues.

Insert Figure 29 about here

5.5 Checking

The total time to complete the solution was not measured because another developing strategy actually increased the solution time. As a result of computer debugging experience, students more frequently checked the directions after making a change. This task provided no opportunity to re-run the directions after making a change as is possible in computer programming; however, the students attempted to re-simulate the effect of the change on their own to test its correctness.

The number of students who read and simulated lines after the initial bug was identified increased steadily across tests for all test types, $\chi^2(4) = 13.77$, $p < .05$. (See Figure 30.) This checking strategy is largely responsible for the increase in correct responses since checking the fix after it has been made can lead to discovering an incorrect fix. Students rarely discovered incorrect fixes on the pre-test because they almost never simulated the effects of the fix they made. One thing that students have clearly learned from debugging experience is that the fix may be wrong or may make things worse. The production system model always instructs the user to recheck the program once a fix has been made. Even though retesting is not easy for debugging non-computer directions, students demonstrated that they knew a very important goal: to check the fixes.

Insert Figure 30 about here

5.6 Transfer Summary

To summarize, the focused search strategies students acquired in the LOGO context were transfered to this non-programming context. Their increasingly focused search and increased checking after making a fix resulted in decreased search time and greater accuracy of fixes. These effects cannot be attributed to either of the alternative hypotheses suggested in Chapter 3. The effect cannot be attributed purely to maturation because the older half of the subjects were not more accurate and did not demonstrate better search or checking strategies than the younger subjects. The older subjects were significantly faster than the younger subjects on the pre-test ($t\ [64] = 3.418$, $p < .05$), but there was no mid-test or post-test difference. The effect of test familiarity cannot be directly tested because of the difference in difficulty of the three test types and the absence of counterbalancing within test session. Students always took more time and were less accurate on the traveling tests, which were always given last in the session. The other two test types were nearly equal in difficulty. Considering only those two test types, the differences between test sessions were significant while the differences within test sessions were not. The best evidence, however, that the quantitative effects are really the result of transfer is that the search strategies were evaluated and shown to shift toward more focused search on both the computer and non-computer debugging tasks.

6. Conclusions

The dream of finding transfer of high-level thinking skills from computer programming has rapidly become a nightmare of mixed results because of failures to match the skills students learned with the skills target tasks required as well as because of poor methodology. The goal of this research was to demonstrate the possibility of achieving high-level transfer of debugging skills when the relevant skills have been appropriately specified, actually learned in the source domain, and directly useful in the target domain. Thus, the approach used for this research had three phases. First, a model of debugging skills was instantiated as a production system to specify the debugging skills students would need to learn in order to debug well. Since students did not learn these skills spontaneously in a pilot study, a curriculum was designed to teach students the model's knowledge and goal structure explicitly. Students' debugging skills were assessed during each of two mini-courses and their performance in the second mini-course was compared to the performance of students who took that mini-course first. Debugging skills were also assessed on non-programming tests designed specifically to require similar skills to program debugging. Thus, the transferability of debugging skills from one LOGO programming domain to a second LOGO programming domain and to non-programming domains could be assessed.

6.1 Support for the thesis

The thesis that this dissertation attempts to test can be stated simply as follows:

- Children can learn high-level thinking skills from computer programming if the component skills are precisely specified and taught directly.
- Once the skills have been learned, they are available for transfer provided they are recognizable as relevant to the target task.

The most direct support for the thesis is that the students who were taught debugging explicitly in the context of a LOGO course (either graphics or list-processing) did learn debugging skills and did transfer them to a second LOGO mini-course and to non-programming tasks requiring debugging of directions. Their acquired focused search strategies for debugging LOGO programs which transfered from one LOGO context to the other and from the LOGO context to a non-programming context. The debugging skills learned in the first mini-course were easily recognizable as relevant to the second mini-course; in fact, the comparison shows a large savings. Also, the students requested the

debugging charts even before debugging was introduced in the second mini-course so they apparently recognized the potential usefulness of the skills they had used before. Debugging is composed of many sub-skills which may be learned and transfered to different degrees depending on the particular experience and tasks. Only the goal structure of debugging was directly relevant to the non-programming task used in this study (and none of the programming and editing skills were predicted to be relevant). Structural clues were relevant to the extent that the directions contained headings recognizably similar to subprograms. The strategy shifts observed on the transfer tests indicated increasing focus on the part of the directions containing the bug (use of structural clues), decreasing tendency to simulate irrelevant directions, and increasing attempts to retest the directions after the fix was made.

The learning of transferable debugging skills in this study was achieved by adding only a small amount (1/2 to 1 hour) of explicit instruction. Once the desired skills had been specified by writing the computer simulation model, the curriculum could be designed and implemented easily. Only gentle reminders to practice the debugging skills, the presence of the debugging posters, and the experimenter demonstrating the skills were necessary on a continual basis after the initial explicit instruction. It is not necessary to teach a whole course on debugging in order for the students to learn transferable skills; it is important, however, for the skills and the knowledge they require to be made explicit and to be used. Perkins and Martin (1986) made a similar suggestion for remedying students early difficulties with computer programming. They are currently testing the hypothesis that students' fragile knowledge can be bolstered by teaching them to use metacognitive strategies to guide their behavior. As for the debugging strategies, the explicit instruction would be minimal, but the continued practice would be essential.

The learning and transfer of debugging skills did not depend on the children first becoming good programmers. Since the good debugging strategies bypass the need to interpret every command (which the pilot study showed children did poorly), debugging can proceed without great programming skills (though some of the most difficult bugs to correct were also areas of difficulty in programming). Being able to debug before being able to program well is especially important since more bugs are generated by novices. Kessler and Anderson (1986) agree that debugging can be learned prior to programming; they taught LISP students to debug simple functions before having experience programming them. Bassok and Holyoak (1986) also suggest that "interdomain transfer need not necessarily

await achievement of exceptional expertise in the base domain"; in other words, the problem-solving procedures can transfer to a target domain even when they are not always executed flawlessly in the source domain. This possibility applies directly to the transfer of the debugging goal structure even though students did not know all of the discrepancy-bug mappings for programming yet.

In addition to these positive results, other results suggest that the converse of the claim is also true:

- There will be no transfer of available skills that are not relevant to the target task.
- Without direct instruction, high-level skills are not likely to be learned and, therefore, cannot transfer.

The results of the programming and editing tests show that the students did improve in both areas. These skills were useful in the second mini-course and resulted in a large savings. However, the transfer tasks were not designed so that these skills would be useful. Detailed analysis of programming and editing skills could reveal tasks to which these skills might transfer; however, that issue was not the focus of this study. Other research has shown that programming skills do transfer to relevant tasks such as angle estimation (Garlick, 1984) and figure comparison (Clements and Gullo, 1984). Similarly, while the goal structure and structural clues for debugging transfered to the non-programming task, the discrepancy-bug mappings for programming were not applicable and, therefore, did not transfer. In fact, only a subset of the discrepancy-bug mappings were relevant for transfer from graphics to list-processing or vice versa.

The LOGO skill tests also revealed that other high-level skills did not develop as debugging skill did. The most difficult problem students had was with decomposing the programming problem in order to plan an appropriate program structure. Thus, they had difficulty deciding which program schemas to use and how to direct the flow of control. In fact, the level of program structure was low unless the program goal required it and the experimenter offered many suggestions. The students also failed to acquire debugging strategies that were not included in the specific instruction, such as documenting programs and using print statements as markers while testing programs. Students also rarely used external information (such as symmetry clues from other parts of the program or bulletin board reference) to ease their problem solving.

These same students participated in two other studies. The results support the notion that skills that have not been learned in the source domain cannot be expected to transfer. Yant (1986) compared these students' planning abilities to those of a control group with no LOGO experience. Her task involved directing a robot to tidy a room under various constraints (a one-armed robot with lots of gas versus a three-armed robot with limited gas). She found that both the LOGO and the no-LOGO groups used poor planning strategies. The LOGO group was better able to give a list of directions without getting lost, however. These results are not surprising since the LOGO group was already described as having difficulty planning and since giving directions is clearly a skill with which LOGO students have vast experience.

Dunbar and Klahr (1986) found that these same LOGO students were no better or worse than students who had not had LOGO at discovering how to use the REPEAT key on a programmable toy (Big Trak). The fact that LOGO students had no advantage is not surprising since students do not have to discover how individual commands operate in a LOGO environment; they are told what arguments are required and what effect they have. However, the LOGO students were expected to be at a disadvantage since the use of REPEAT in a LOGO environment is a misleading analogy for the Big Trak.

6.2 Possibilities for strengthening the evidence

Being able to document the acquisition and transfer of specific debugging skills is strong evidence in support of the thesis. However, the evidence could be stronger in the following ways.

First, the results should be replicated with other students at other schools with other teachers to show that the effects are general. Also, the design could be improved by having various control groups either without LOGO experience or in a LOGO class without explicit debugging instruction. A study including a comparison of two groups, one with computer first semester and study hall second and one with the reverse schedule is already planned for the coming fall. A second planned study includes LOGO classes from the same grades at the same school with and without explicit instruction in debugging.

The tie between the learning and transfer of debugging skills could also be made more direct. First, assuming that students could be taught to give better protocols, individualized testing could be used so that the learning path for individual students could be compared to

each student's level of transfer. A strong correlation of individual learning and transfer scores would be strong support for the thesis. In addition, the learning path could be traced better by counterbalancing the tests within each mini-course.

Another important extension would be to strengthen the transfer effect. Some of the LOGO students in this study did not shift strategies as a result of the debugging instruction. Perhaps more instruction, examples, and/or practice would facilitate the shift for those students by insuring complete and abstract learning (Smith, 1986; Bassok and Holyoak, 1986). Several researchers have suggested that giving a hint that a previous strategy is relevant would improve transfer (Gentner and Gentner, 1983; Gick and Holyoak, 1983; and Holyoak and Koh, 1986). Transfer might be improved by giving students a hint to use their debugging strategies. When discussing whether exposure to the LogoWriter microworld would lead to better writing skills in general, Papert (1986) commented,

> If you see transfer as an automatic process that needs no encouragement from you [the teacher], it may or may not. But I am convinced that your imagination as a teacher will show you how to use the LogoWriter programming as a transition to pure writing.

Yet another possibility is to show that the approach works for other high-level skills too. Problem decomposition and structured programming have been shown to be very difficult for children yet are crucial for good programming (and are helpful for good debugging). Fisher (1986) is developing a model of how expert programmers decompose a problem into a programming plan. Such a model would be useful for designing explicit instruction in much the same way as the Carver and Klahr (1986) debugging model was. Again, skill acquisition could be assessed and transfer to tasks with recognizable similarity measured.

Finally, there is strength in predictability. The model must be developed to the extent that it can predict the amount of transfer to various tasks depending on the relevance of the learned skills. For example, we found differences between transfer tasks where the directions had subprogram-like heading and where they did not. However, these differences were not tested directly with controlled contexts. The goal must be to design transfer items with predictable differences and then to test those predictions.

6.3 Applying the findings and the approach

Attempts must also be made to apply these claims directly to educators, both those teaching programming and those teaching other subjects. Curriculum materials based on the current findings could be developed for use in programming courses, and the possible transfer effects could be specified in terms of the educational objectives actually used in the schools. This dissertation showed that students could learn focused search strategies in a LOGO context and transfer them to a non-programming context. Such skills might also be transferable to other complex search tasks typically encountered in school, such as locating a particular entry in a dictionary or other reference book and scaning text to find a particular piece of information to answer a question.

For more general applicability, the analysis/instruction/assessment approach used here must be described thoroughly enough that educators and researchers in other fields can apply it. This dissertation research has shown that using a performance theory to guide instruction and assessment can lead to realizing the dream of students learning transferable debugging skills. Yet, reality can only conform to the dream as researchers continue to explore the nature of transfer and educators begin to utilize the findings in the classroom.

Table 1: Discrepancy-Bug Mappings in the GRAPES Model

DISCREPANCY	EXAMPLE DESCRIPTION	BUGGY COMPONENT	SPECIFIC BUG
Orientation	"this is going over here instead of down"	Angle	LT n or RT n
Size	"that line - it's too long"	Distance	FD n or BK n
Spread	"these are too close together"	Angle or Distance	LT n or RT n or FD n or BK n
Location	"this is supposed to be in the middle"	Distance	FD n or BK n
Extent	"lots too many squares"	Iteration or Recursion stop or Recursion interval	REPEAT n IF :x = n THEN STOP NAME :x +- n
Extra part	"it drew a line there"	Pen position or Program call	PU omitted or Extra call
Wrong part	"it drew corn instead of a stalk"	Program call	Switched call
Missing part	"I wanted a box there"	Pen position or Program call	PD omitted or Call omitted
Print variable	"it printed 'score' instead of the number"	Punctuation	Quoted variable
Not matching	"I put the right answer and it marked it wrong"	Nesting	READLIST or READWORD
Wrong value	"it printed the number instead of the place"	Variable name	Wrong variable name
How to	ERROR MESSAGE	Punctuation	Missing punctuation
What to	ERROR MESSAGE	Command	Missing command or Missing prenthesis
No value	ERROR MESSAGE	Initialization	MAKE "name value or No parameter
Don't know	"this mess"	?	?

Table 2: Example Trace of the GRAPES Model with High Information

```
[*] start
top-goal: test FARM
RUN
Run the program FARM.                                          -->ok
MATCH
Did the outcome match the plan? [yes or no]?                   -->no
|  1) test-1.
|  goal-1: evaluate FARM
|  |  2) evaluate-2.
|  |  goal-2: describe FARM
CONTRAST
What type of discrepancy is there?
[graphics or lists]                                            -->graphics
CONTRAST
Did you get an error message?                                  -->no
|  |  |  3) describe-1.
|  |  |  goal-6: describe FARM
CONTRAST
What is a discrepancy between the plan and outcome?
[orientation, size, spread, location, extent,
extrapart, wrongpart, missingpart, or ?]                       -->wrongpart
|  |  |  |  4) describe-2.
|  |  |  |  goal-7: propose FARM
The program is probably calling the wrong subprogram.
EXAMINE
What is the subprogram that actually ran?                      -->CORN
|  |  |  |  |  5) propose-13.
|  |  |  |  Goal successful.
|  |  |  Goal successful.
|  |  Goal successful.
|  |  goal-3: represent FARM
RECALL and EXAMINE
Does the FARM program have subprograms?                        -->?
|  |  |  6) represent-2.
|  |  |  goal-8: specify FARM
EXAMINE
Is the bug in a REPEAT or IF statement?                        -->yes
|  |  |  |  7) specify-3.
|  |  |  |  goal-9: specify FARM
EXAMINE
Which?                                                         -->REPEAT
|  |  |  |  |  8) specify-4.
|  |  |  |  Goal successful.
|  |  |  Goal successful.
|  |  Goal successful.
|  |  goal-4: find FARM
|  |  |  9) find-8.
|  |  |  goal-10: find FARM
The bug is (CORN) in FARM.
((PU) (SETPOS (130) (50) (B)) (PD) (LT (90)) (FD (260)) (BK (10)) (RT (90))
(REPEAT (4) xxx (CORN) (RT (90)) (FD (25)) (LT (90)) (B)) (PU) (FD (10)) (PD)
(SILO) (PU) (BK (50)) (PD) (RT (90)) (BARN))
```

```
|  |  |  |  10) find-6.
|  |  |  Goal successful.
|  |  Goal successful.
|  |  goal-5: change FARM
GENERATE
How should the fix be made?
[change, delete, or insert]                                    -->change
|  |  |  11) change-1.
|  |  |  goal-11: change FARM
GENERATE
What should the (CORN) have been?                              -->(STALK)
|  |  |  |  12) change-2.
|  |  |  |  goal-12: replace FARM
ENTER, SKIP, DELETE, INSERT
((PU) (SETPOS (130) (50) (B)) (PD) (LT (90)) (FD (260)) (BK (10)) (RT (90))
(REPEAT (4) (STALK) (RT (90)) (FD (25)) (LT (90)) (B)) (PU) (FD (10)) (PD)
(SILO) (PU) (BK (50)) (PD) (RT (90)) (BARN))
|  |  |  |  |  13) replace-1.
|  |  |  |  |  goal-13: test FARM
RUN
Run the program FARM.                                          -->ok
MATCH
Did the correction fix the problem?                            -->yes
|  |  |  |  |  |  14) test-2.
|  |  |  |  |  |  goal-14: evaluate FARM
|  |  |  |  |  |  |  15) evaluate-3.
|  |  |  |  |  |  |  goal-15: test FARM
MATCH
Did the outcome match the plan? [yes or no]?                   -->yes
|  |  |  |  |  |  |  |  16) test-3.
|  |  |  |  |  |  |  |  goal-16: evaluate FARM
|  |  |  |  |  |  |  |  |  17) evaluate-1.
|  |  |  |  |  |  |  |  Goal successful.
|  |  |  |  |  |  |  Goal successful.
|  |  |  |  |  |  Goal successful.
|  |  |  |  |  Goal successful.
|  |  |  |  Goal successful.
|  |  |  Goal successful.
|  |  Goal successful.
|  Goal successful.
Goal successful.
-- Top Goal Successful.
No Productions Applicable.
```

Table 3: Example Trace of the GRAPES Model with Low Information

```
[*]start

top-goal: test FARM
RUN
Run the program FARM.                                          -->ok
MATCH
Did the outcome match the plan? [yes or no]?                   -->no
|  1) test-1.
|  goal-1: evaluate FARM
|  |  2) evaluate-2.
|  |  goal-2: describe FARM
CONTRAST
What type of discrepancy is there?
[graphics or lists]                                            -->graphics
CONTRAST
Did you get an error message?                                  -->no
|  |  |  3) describe-1.
|  |  |  goal-6: describe FARM
CONTRAST
What is a discrepancy between the plan and outcome?
[orientation, size, spread, location, extent,
extrapart, wrongpart, missingpart, or ?]                       -->?
|  |  |  |  4) describe-2.
|  |  |  |  goal-7: propose FARM
|  |  |  |  |  5) propose-1.
|  |  |  |  Goal successful.
|  |  |  Goal successful.
|  |  Goal successful.
|  |  goal-3: represent FARM
RECALL and EXAMINE
Does the FARM program have subprograms?                        -->?
|  |  |  6) represent-2.
|  |  |  goal-8: specify FARM
EXAMINE
Is the bug in a REPEAT or IF statement?                        -->no
|  |  |  |  7) specify-3.
|  |  |  |  goal-9: specify FARM
EXAMINE
Is there any command that could be used as a marker?           -->no
|  |  |  |  |  8) specify-5.
|  |  |  |  |  goal-10: specify FARM
|  |  |  |  |  |  9) specify-7.
|  |  |  |  |  Goal successful.
|  |  |  |  Goal successful.
|  |  |  Goal successful.
|  |  Goal successful.
|  |  goal-4: find FARM
|  |  |  10) find-1.
|  |  |  goal-11: find FARM
|  |  |  |  11) find-2.
|  |  |  |  goal-12: interpret FARM
READ and INTERPRET
```

```
(PU lifts the turtle 's pen so that no trace will be left when it moves.)
MATCH
Is PU the right command? [yes or no]                          -->yes
|  |  |  |  |  12) interpret-4.
|  |  |  |  |  goal-13: check FARM
|  |  |  |  |  |  13) check-2.
|  |  |  |  Goal fails.
|  |  |  |  goal-12: interpret FARM
READ and INTERPRET
(SETPOS places the turtle at the specified x y coordinates.)
MATCH
Is SETPOS the right command? [yes or no]                      -->yes
|  |  |  |  |  14) interpret-4.
|  |  |  |  |  goal-14: check FARM
|  |  |  |  |  |  15) check-2.
|  |  |  |  Goal fails.
|  |  |  |  goal-12: interpret FARM
MATCH
Is 130 the right number? [yes or no]                          -->yes
|  |  |  |  |  16) interpret-7.
|  |  |  |  |  goal-15: check FARM
|  |  |  |  |  |  17) check-2.
|  |  |  |  Goal fails.
|  |  |  |  goal-12: interpret FARM
MATCH
Is 50 the right number? [yes or no]                           -->yes
|  |  |  |  |  18) interpret-7.
|  |  |  |  |  goal-16: check FARM
|  |  |  |  |  |  19) check-2.
|  |  |  |  Goal fails.
|  |  |  |  goal-12: interpret FARM
MATCH
Is (Bracket) the right punctuation? [yes or no]               -->yes
|  |  |  |  |  20) interpret-5.
|  |  |  |  |  goal-17: check FARM
|  |  |  |  |  |  21) check-2.
|  |  |  |  Goal fails.
|  |  |  |  goal-12: interpret FARM
READ and INTERPRET
(PD puts the turtle 's pen down so that a trace will be left when it moves.)
MATCH
Is PD the right command? [yes or no]                          -->yes
|  |  |  |  |  22) interpret-4.
|  |  |  |  |  goal-18: check FARM
|  |  |  |  |  |  23) check-2.
|  |  |  |  Goal fails.
|  |  |  |  goal-12: interpret FARM
READ and INTERPRET
(LT turns the turtle to the left a certain number of degrees.)
MATCH
Is LT the right command? [yes or no]                          -->yes
|  |  |  |  |  24) interpret-4.
|  |  |  |  |  goal-19: check FARM
|  |  |  |  |  |  25) check-2.
```

```
| | | | Goal fails.
| | | | goal-12: interpret FARM
MATCH
Is 90 the right number? [yes or no]                          -->yes
| | | | | 26) interpret-7.
| | | | | goal-20: check FARM
| | | | | | 27) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
READ and INTERPRET
(FD moves the turtle forward a certain number of turtle steps.)
MATCH
Is FD the right command? [yes or no]                         -->yes
| | | | | 28) interpret-4.
| | | | | goal-21: check FARM
| | | | | | 29) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
MATCH
Is 260 the right number? [yes or no]                         -->yes
| | | | | 30) interpret-7.
| | | | | goal-22: check FARM
| | | | | | 31) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
READ and INTERPRET
(BK moves the turtle backward a certain number of turtle steps.)
MATCH
Is BK the right command? [yes or no]                         -->yes
| | | | | 32) interpret-4.
| | | | | goal-23: check FARM
| | | | | | 33) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
MATCH
Is 10 the right number? [yes or no]                          -->yes
| | | | | 34) interpret-7.
| | | | | goal-24: check FARM
| | | | | | 35) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
READ and INTERPRET
(RT turns the turtle to the right a certain number of degrees.)
MATCH
Is RT the right command? [yes or no]                         -->yes
| | | | | 36) interpret-4.
| | | | | goal-25: check FARM
| | | | | | 37) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
MATCH
Is 90 the right number? [yes or no]                          -->yes
| | | | | 38) interpret-7.
| | | | | goal-26: check FARM
```

```
| | | | | | 39) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
READ and INTERPRET
(REPEAT executes a list of commands a specified number of times.)
MATCH
Is REPEAT the right command? [yes or no]                    -->yes
| | | | | 40) interpret-4.
| | | | | goal-27: check FARM
| | | | | | 41) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
MATCH
Is 4 the right number? [yes or no]                          -->yes
| | | | | 42) interpret-7.
| | | | | goal-28: check FARM
| | | | | | 43) check-2.
| | | | Goal fails.
| | | | goal-12: interpret FARM
Was (CORN) the appropriate subprogram call here?            -->no
| | | | | 44) interpret-1.
| | | | | goal-29: check FARM
The bug is the (CORN) in FARM.
((PU) (SETPOS (130) (50) (B)) (PD) (LT (90)) (FD (260)) (BK (10)) (RT (90))
(REPEAT (4) xxx (CORN) (RT (90)) (FD (25)) (LT (90)) (B)) (PU) (FD (10)) (PD)
(SILO) (PU) (BK (50)) (PD) (RT (90)) (BARN))
| | | | | | 45) check-1.
| | | | | Goal successful.
| | | | Goal successful.
| | | Goal successful.
| | Goal successful.
| | goal-5: change FARM
GENERATE
How should the fix be made?
[change, delete, or insert]                                 -->change
| | | 46) change-1.
| | | goal-30: change FARM
GENERATE
What should the (CORN) have been?                           -->(STALK)
| | | | 47) change-2.
| | | | goal-31: replace FARM
ENTER, SKIP, DELETE, INSERT
((PU) (SETPOS (130) (50) (B)) (PD) (LT (90)) (FD (260)) (BK (10)) (RT (90))
(REPEAT (4) (STALK) (RT (90)) (FD (25)) (LT (90)) (B)) (PU) (FD (10)) (PD)
(SILO) (PU) (BK (50)) (PD) (RT (90)) (BARN))
| | | | | 48) replace-1.
| | | | | goal-32: test FARM
RUN
Run the program FARM.                                       -->ok
MATCH
Did the correction fix the problem?                         -->yes
| | | | | | 49) test-2.
| | | | | | goal-33: evaluate FARM
| | | | | | | 50) evaluate-3.
```

```
| | | | | | | goal-34: test FARM
MATCH
Did the outcome match the plan? [yes or no]?                    -->yes
| | | | | | | | 51) test-3.
| | | | | | | | goal-35: evaluate FARM
| | | | | | | | | 52) evaluate-1.
| | | | | | | | Goal successful.
| | | | | | | Goal successful.
| | | | | | Goal successful.
| | | | | Goal successful.
| | | | Goal successful.
| | | Goal successful.
| | Goal successful.
| Goal successful.
Goal successful.
-- Top Goal Successful.
No Productions Applicable.
```

Table 4: Description of the Subjects

Group	Name Code	Grade	Age[1]	National Percentile Ranks[2] Reading	Math	Language	Computer
A	AF	3	8;2	85	95	79	
	MJ	3	8;3	87	99	68	
	ER	3	8;5	98	99	84	
	KK	3	8;9	74	95	73	Commodore
	BR	4	9;6	84	91	80	IBM PC
	PE	4	9;7	76	87	67	Apple IIc
	AG	5	9;9	98	99	98	Commodore
	AC	4	9;10	79	87	53	TI
	AP	5	10;7	78	73	79	Commodore
	DC	5	10;10	96	98	98	IBM PC
	BS	5	11;0	95	81	96	
	Ave.	4	9;6	86	91	80	7/11
B	MF	3	8;4	77	94	68	
	TH	3	8;7	46	52	41	Apple IIc
	CL	3	8;7	65	71	53	IBM PC
	CH	5	8;10	97	96	99	IBM PC
	BC	4	9;3	89	99	86	IBM PC
	CP	4	9;10	64	72	53	
	RM	5	10;2	93	94	86	
	MH	5	10;6	81	38	51	Apple IIc
	BH	5	10;7	66	71	28	Apple IIc
	KB	6	11;1	76	89	82	
	RG	6	11;8	93	94	78	
	Ave.	4.5	9;9	77	79	66	6/11

[1] At the beginning of the course.

[2] Since the subjects span several grades, the scores are not comparable across tests. The 3rd graders took the Stanford Achievement Test, Primary 3, Form E, Complete Battery. The 4th and 5th graders took the Stanford Achievement Test, Intermediate 1, Form E, Complete Battery. The 6th graders took the Metropolitan Achievement Test, Intermediate Form KS, Complete Battery.

Table 5: Sequence of Graphics Lessons

Each graphics lesson was approximately one hour long.

	Graphics (first Mini)	Graphics (second Mini)
1	demonstration	
2	basic commands	1 basic commands
3	interactive project	
4	name projects	2 name projects/SET...
5	REPEAT	3 REPEAT
6	curves	4 curves
7	**PROGRAM TEST 1**	5 **PROGRAM TEST 1**
8	subprograms/farm projects	6 subprograms/ farm projects
9	**DEBUG TEST 1**	7 **DEBUG TEST 1**
10	original projects	8 original projects
11	**EDIT TEST 1**	9 **EDIT TEST 1**
12	original projects	
13	debugging	10 debugging
14	variable shape programs	11 variable shape programs
15	more variables	12 more variables
16	**PROGRAM TEST 2**	13 **PROGRAM TEST 2**
17	seashore projects	14 seashore projects
18	**DEBUG TEST 2**	15 **DEBUG TEST 2**
19	original projects	16 recursion
20	**EDIT TEST 2**	17 **EDIT TEST 2**
21	recursion	
22	more recursion	18 more recursion
23	**PROGRAM TEST 3**	19 **PROGRAM TEST 3**
24	garden projects	20 garden projects
25	**DEBUG TEST 3**	21 **DEBUG TEST 3**
26	original projects	22 original projects
27	**EDIT TEST 3**	23 **EDIT TEST 3**

Table 6: Sequence of List-processing Lessons

Each list-processing lesson was approximately one hour long.

Lists (first Mini)		Lists (second Mini)	
1	demonstration		
2	words and lists		
3	PRINT, MAKE, and IF	1	PRINT, MAKE, and IF
4	interview projects	2	interview projects
5	printing	3	printing
6	finish interviews	4	original projects
7	**PROGRAM TEST 1**	5	**PROGRAM TEST 1**
8	subprograms	6	subprograms
9	**DEBUG TEST 1**	7	**DEBUG TEST 1**
10	madlib projects	8	madlib projects
11	**EDIT TEST 1**	9	**EDIT TEST 1**
12	quiz projects		
13	debugging	10	debugging
14	variables/recursion	11	variables
15	variables/recursion	12	variables/recursion
16	**PROGRAM TEST 2**	13	**PROGRAM TEST 2**
17	unscrambler projects	14	unscrambler projects
18	**DEBUG TEST 2**	15	**DEBUG TEST 2**
19	original projects	16	RANDOM (number guessing)
20	**EDIT TEST 2**	17	**EDIT TEST 2**
21	RANDOM (number guessing)		
22	ITEM/COUNT (silly stories)	18	ITEM/COUNT(silly stories)
23	**PROGRAM TEST 3**	19	**PROGRAM TEST 3**
24	poetry projects	20	poetry projects
25	**DEBUG TEST 3**	21	**DEBUG TEST 3**
26	original projects	22	original projects
27	**EDIT TEST 3**	23	**EDIT TEST 3**

Table 7: Buggy Directions for Arranging Furniture

Here are the directions Mrs. Fisher gave to the movers.

To arrange the dining room,
- Center the china cabinet on the west wall.
- Place the silver cabinet in the south-east corner.
- Put the table in the center of the room.
- Arrange the 6 chairs around the table evenly.

To arrange the living room,
- Place the cabinets against the west wall.
- Place one chair in front of each end of the cabinets.
- Place the square table in the north-east corner.
- Put the sofa on the north wall, next to the square table.
- Place another chair on the south wall, across from the sofa.
- Put the coffee table between the two chairs.
- Put the rocker on the east wall, next to the square table.

To arrange the kitchen,
- Put the refrigerator in the north-west corner.
- Put the dishwasher to the right of the refrigerator.
- Put the sink to the right of the dishwasher.
- Put the stove to the right of the sink.
- Place the counter next to the stove and along the east wall.
- Put the oven along the east wall, next to the counter.
- Place the table in the south-west corner of the room.
- Arrange the 4 chairs around the table evenly.

Change or add one thing to fix Mrs. Fisher's directions.

Figure 1: Sample Graphics Programs

The left-hand panels list the previously stored program, and the right-hand panels illustrate the graphic effects of the commands listed. All drawings start with the turtle positioned at the bottom left of the drawing, oriented to the north. a) LOGO programs to draw a flower of variable size comprised of a line, two leaves, and ten diamonds. b) Recursive LOGO program to draw a series of flowers decreasing in size.

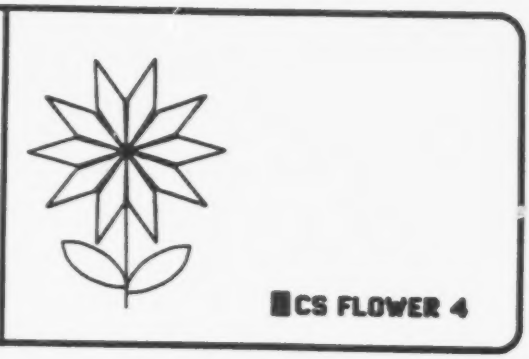
a.

```
TO FLOWER :S
FD 5 LEAF :S FD 5 LT 90 LEAF :S RT 90 FD :S *15
REPEAT 10 [DIAMOND :S * 5 RT 36]
END

TO DIAMOND :C
REPEAT 2 [FD :C RT 48 FD :C RT 132]
END

TO LEAF :S
REPEAT 2 [CURVE :S RT 90]
END

TO CURVE :A
REPEAT 9 [FD :A RT 10]
END
```



b.

```
TO FLOWERS :S
FLOWER :S
IF EQUALP :S 2 [STOP]
PU RT 90 FD :S * 14 LR 90 PD
FLOWERS :S - 1
```

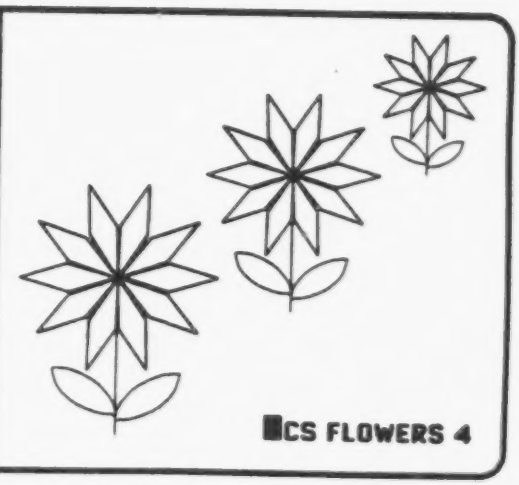


Figure 2: Sample List-processing Programs

The top half of each panel lists the previously stored program, and the bottom half illustrates the interactive effects of the commands listed. The program user typed the inputs beside the rectangular cursor. The intermediate lines show the program's response. a) LOGO program to translate one word into piglatin, b) Recursive LOGO program to translate a sentence into piglatin, c) LOGO program to integrate the other two into a user friendly game.

a.

```
TO PIGGY :W
OUTPUT (WORD BUTFIRST :W FIRST :W "AY)
END
```

```
?PRINT PIGGY "PIG
IGPAY
```

b.

```
TO ALLPIGGY :L
IF EMPTYP :L [OUTPUT []]
OUTPUT SENTENCE PIGGY FIRST :L ALLPIGGY BUTFIRST :L
END
```

```
?PRINT ALLPIGGY [PORKY PIG]
ORKYPAY IGPAY
```

c.

```
TO PIGLATIN
PRINT [PLEASE TYPE A SENTENCE THAT YOU WANT TRANSLATED INTO PIGLATIN]
PRINT ALLPIGGY READLIST
PRINT [WOULD YOU LIKE TO CONTINUE?]
MAKE "Y READWORD
IF EQUALP :Y "NO [PRINT [THANKS FOR PLAYING. HAVE A NICE DAY!]] [PIGLATIN]
END
```

```
?PIGLATIN
PLEASE TYPE A SENTENCE THAT YOU WANT TRANSLATED INTO PIGLATIN
?THIS LITTLE PIGGY WENT TO MARKET
HISTAY ITTLELAY IGGYPAY ENTWAY OTAY ARKETMAY
WOULD YOU LIKE TO CONTINUE?
?YES
PLEASE TYPE A SENTENCE THAT YOU WANT TRANSLATED INTO PIGLATIN
?THIS LITTLE PIGGY WENT HOME
HISTAY ITTLELAY IGGYPAY ENTWAY OMEHAY
WOULD YOU LIKE TO CONTINUE?
?NO
THANKS FOR PLAYING. HAVE A NICE DAY!
```

Figure 3: The Goal Structure of the GRAPES Model

Goal tree for the GRAPES debugging model. [Highlighted goals explained in text.]

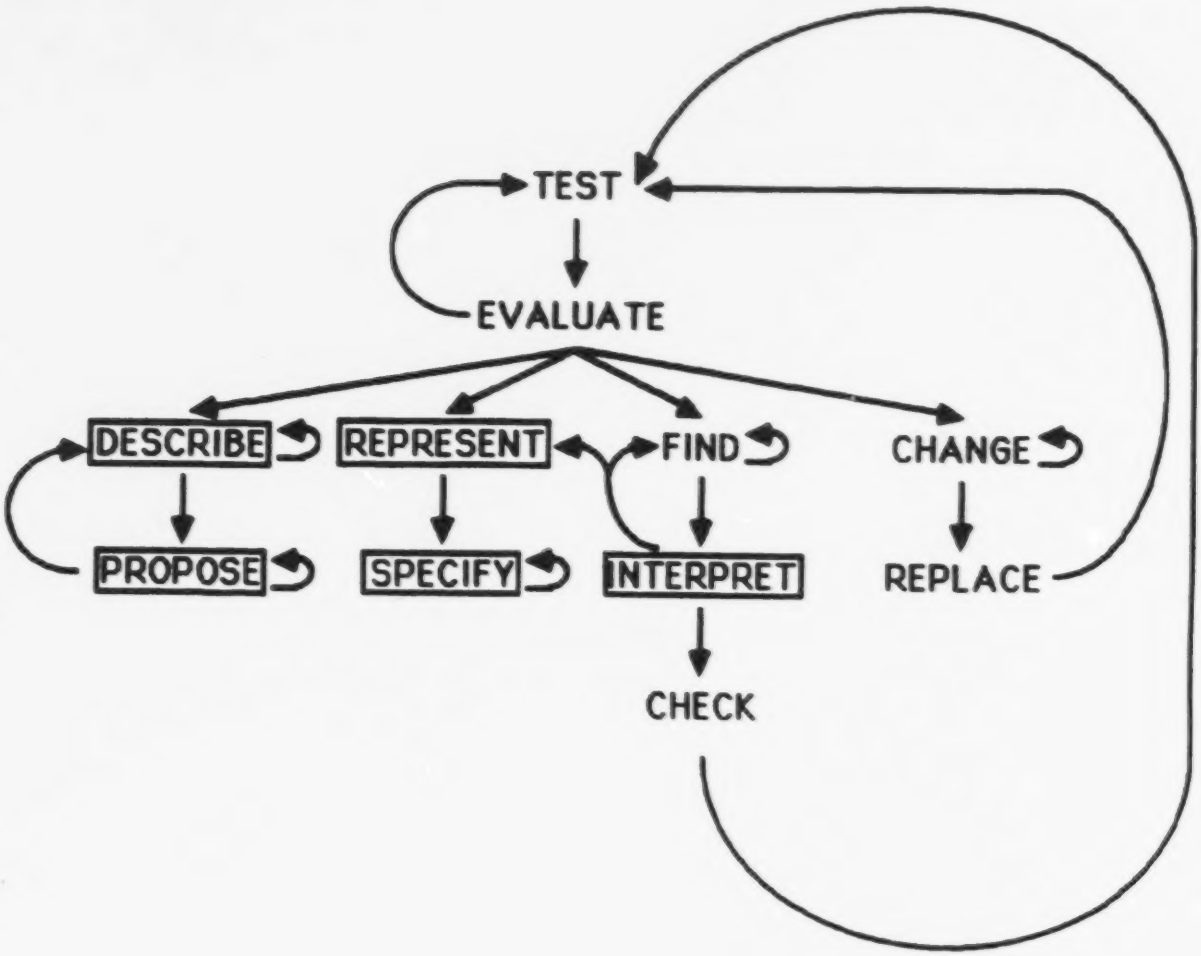
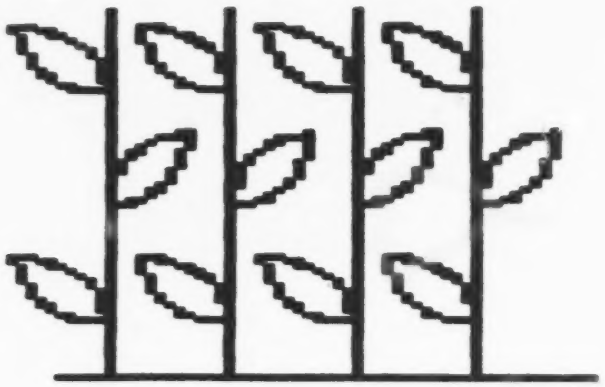


Figure 4: A Sample Debugging Problem

Example of a desired output (a), the actual output (b), and the buggy program that produced the flawed output (c). The bug is the call to the subprogram CORN in FARM. It should be a call to STALK (which in turn calls CORN).

a.



b.

c.

```
TO  FARM
   PU  SETPOS [-20  -50]  PD
   LT 90  FD 110  BK 10  RT 90
   REPEAT 4 [CORN  RT 90  FD 25  LT 90]
END

TO  STALK
   FD 15  LT 90  CORN  RT 90  FD 30  CORN
   FD 50  LT 90  CORN  RT 90  FD 20  BK 95
END

TO  CORN
   REPEAT 2 [REPEAT 9 [FD 3  RT 90]  RT 90]
END
```

Figure 5: High Information Goal Tree

Goal tree generated during high-information debugging of the FARM program. Numbers correspond to the order in which the sub-goals were generated. Parenthetical elements indicate accumulation of information in working memory.

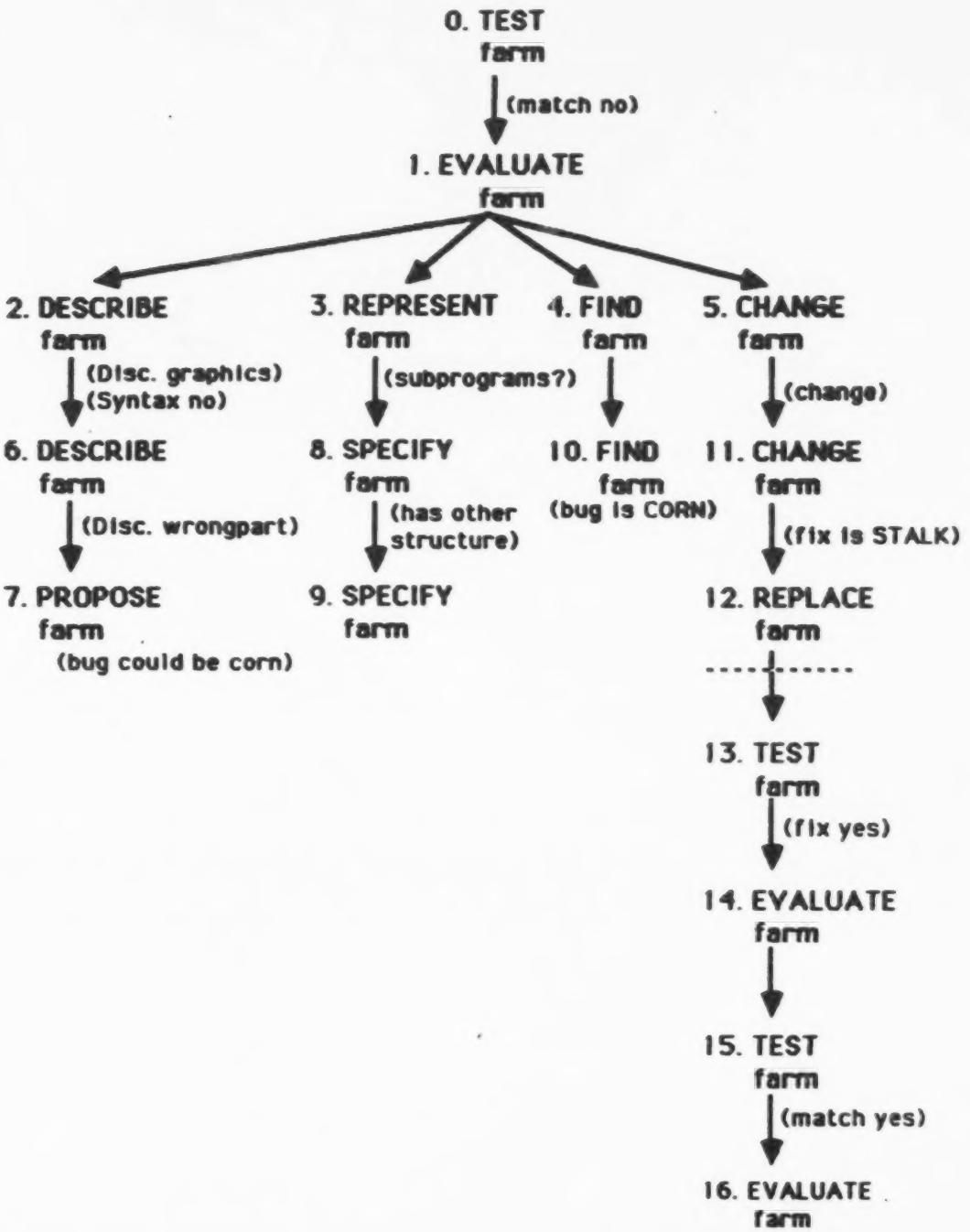
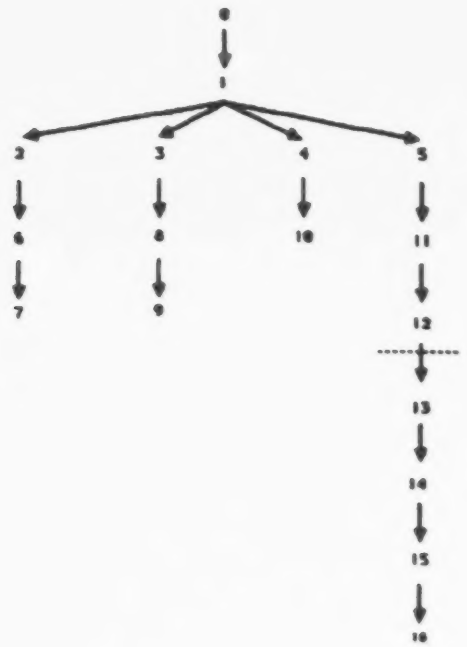


Figure 6: Comparison of High and Low Information Goal Trees

Comparison of schematic goal trees for high-information (a) and low-information (b) traces. Note that (a) is the same goal tree shown in Figure 5, with much of the detail suppressed.

a.



b.

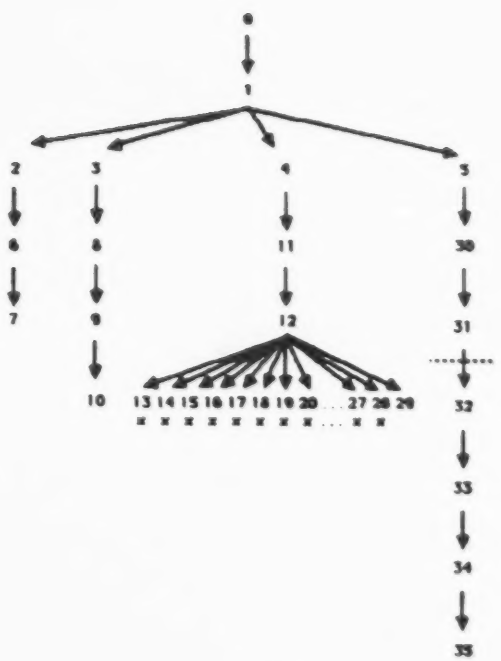
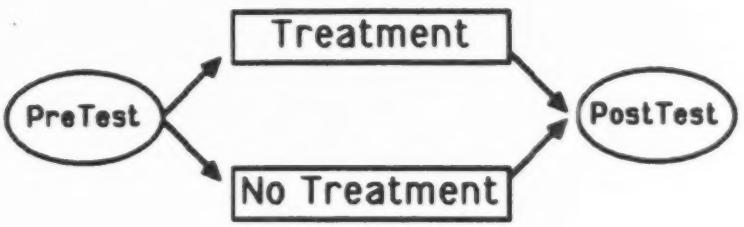


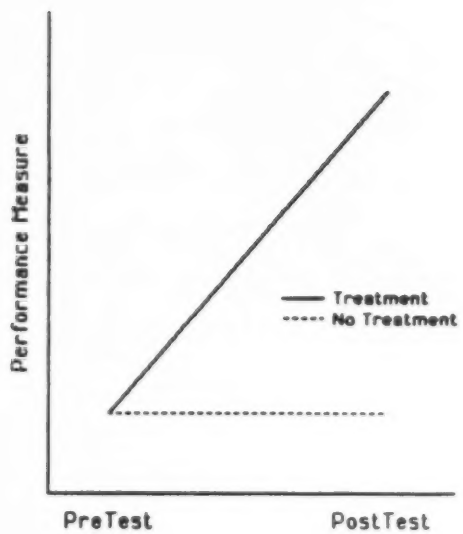
Figure 7: A Pre-test/Post-test Transfer Design.

a) The typical pre-test/post-test transfer design involves testing a group of subjects before and after half of them receive some treatment. b) Greater improvement of the treatment group than the no treatment group from the pre-test to the post-test is evidence of transfer (assuming that there are no confounds in the experiment). c) In the dissertation study, subjects were tested at three times: before, during, and after LOGO experience including debugging instruction. 1, 2, and 3 stand for three types of tests given at each time; a, b, and c represent three different versions which were used so that the tests could be counterbalanced across test time.

a.



b.



c.

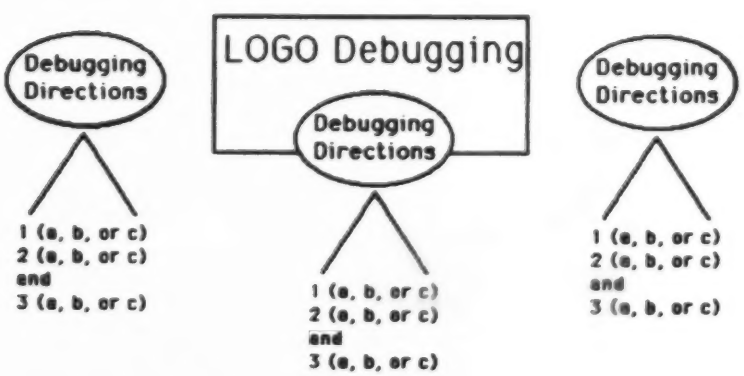
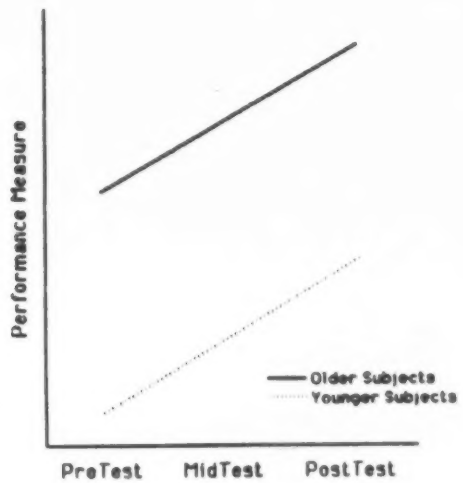


Figure 8: Patterns of Results Suggested by Alternative Hypotheses.

a) One alternative hypothesis is that transfer effects are purely the result of maturation during the time between the tests. If this is the case, then the younger subjects should score lower on the pre-tests than the older students and lower on the post-tests than the older subjects ever were (because the difference in their ages is greater than the test interval). b) A second alternative hypothesis is that transfer effects are the result of practice on the tests. In this case, improvements should be constant across all nine tests (three at each test time) rather than showing increases primarily between test times.

a.



b.

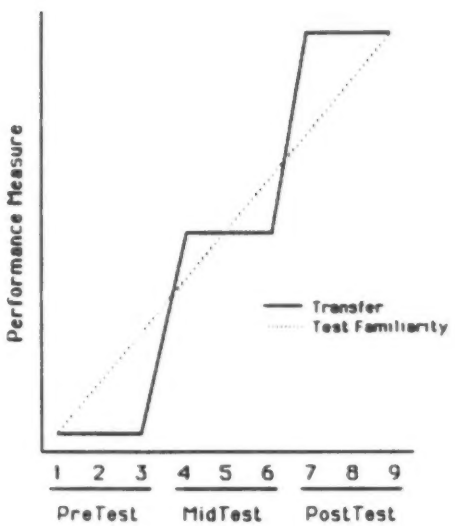
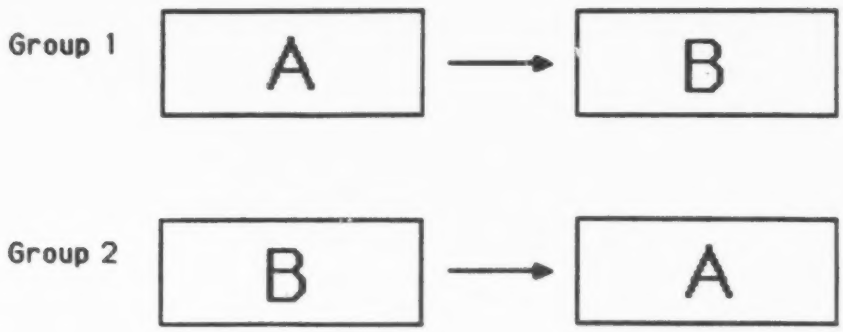


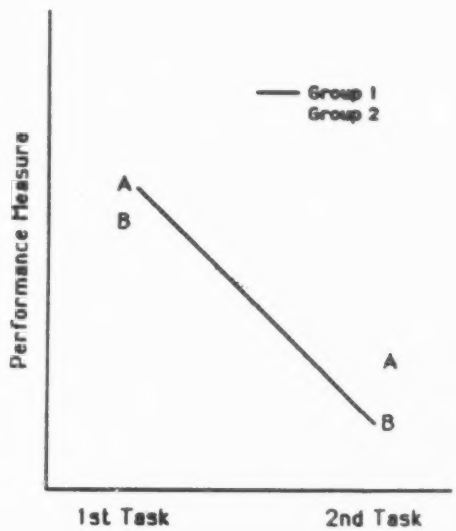
Figure 9: A Savings Transfer Design.

a) A typical savings design involves two groups of subjects doing two tasks in different orders. b) Transfer is indicated by better performance on task A by the group doing task B first and by better performance on task B by the group doing task A first. c) In the dissertation study, the two groups of subjects took two LOGO mini-courses (graphics and list-processing) in different orders. In each course, they took three series of tests, each of which had three items (programming, debugging, and editing).

a.



b.



c.

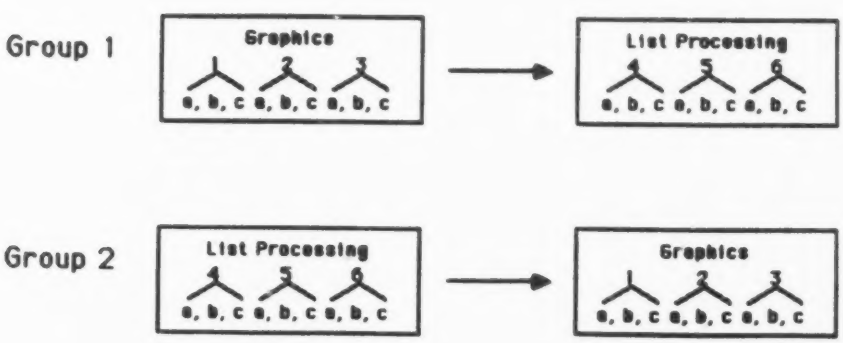


Figure 10: A Combined Pre-test/Post-test and Savings Design.

This figure is a combination of Figures 7c and 9c; it shows the complete design.

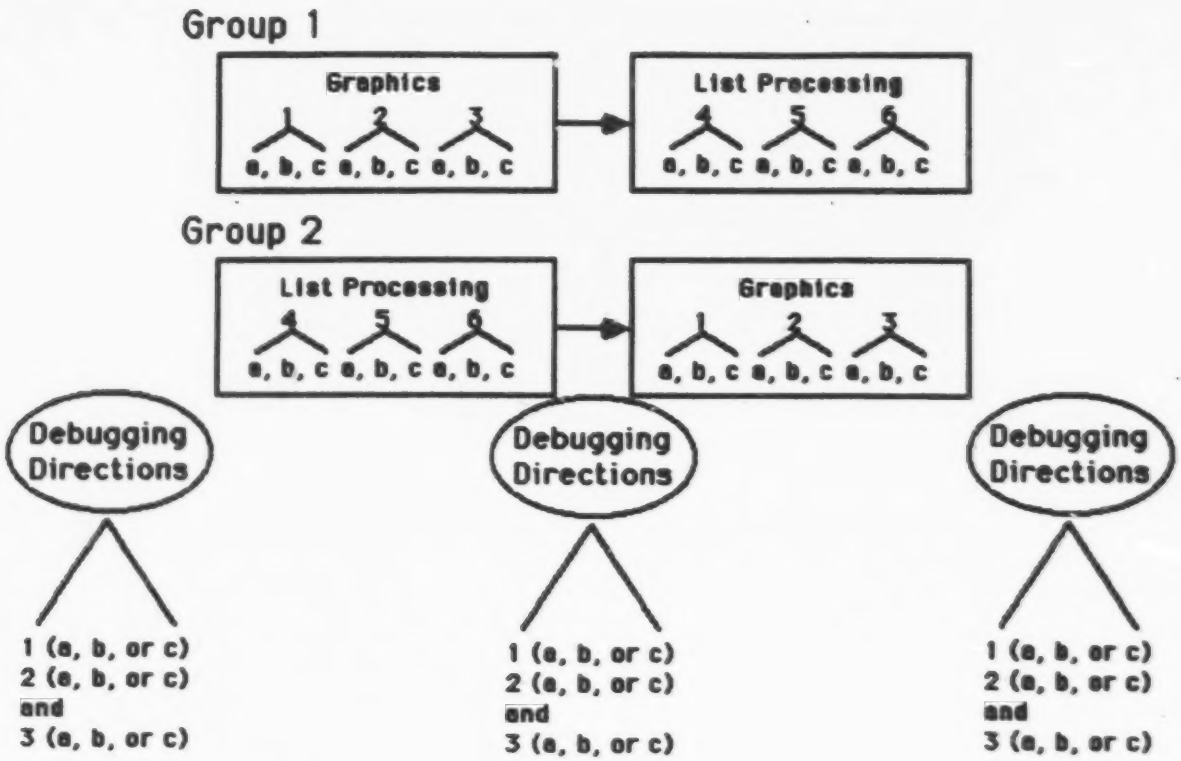


Figure 11: The Model-Based Debugging Instruction

Debugging instruction was derived directly from the model (with changes in wording for the benefit of young students). The step-by-step debugging process taught explicitly in both LOGO mini-courses is represented here in terms of the goal structure of the GRAPES model.

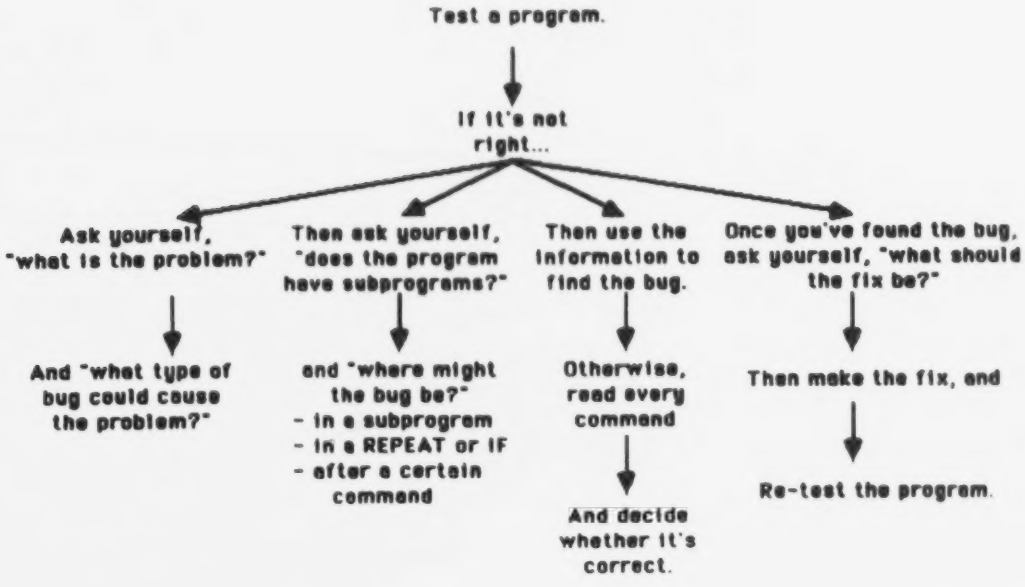

Test a program.
If it's not right...
Ask yourself, "what is the problem?"
Then ask yourself, "does the program have subprograms?"
Then use the information to find the bug.
Once you've found the bug, ask yourself, "what should the fix be?"
And "what type of bug could cause the problem?"
and "where might the bug be?"
- in a subprogram
- in a REPEAT or IF
- after a certain command
Otherwise, read every command
And decide whether it's correct.
Then make the fix, and
Re-test the program.

Figure 12: Planned and Buggy Outcomes for Arranging Furniture

Example of the discrepancy information provided on a transfer test. The bug is the coffee table in the living room. Using this clue would help narrow the search for the buggy direction (see Table 6).

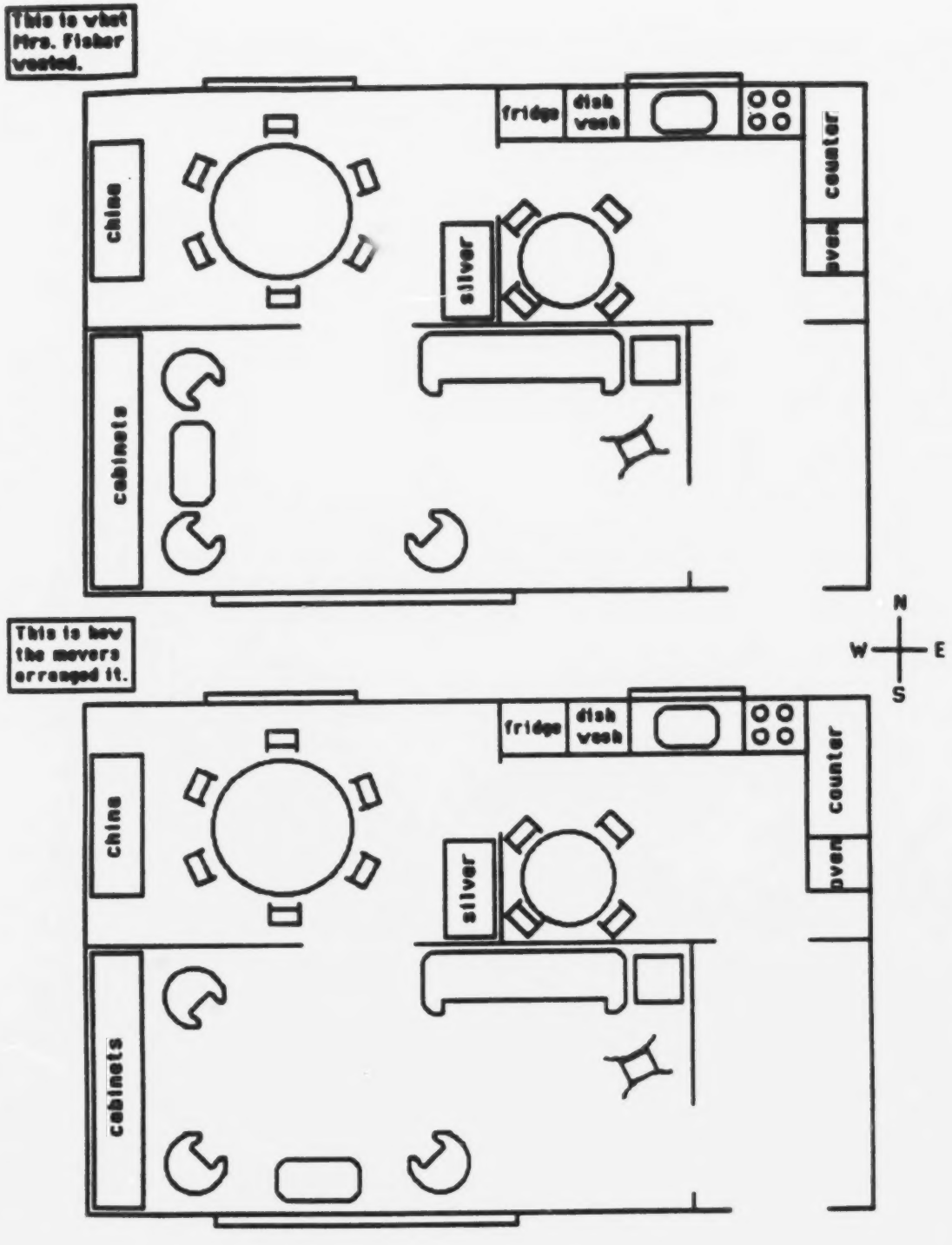


Figure 13: A Sample Debugging Transcript

Example transcript from a debugging test. All transcriptions were done in terms of the model's goal structure. The model's goals are shown in parentheses below each example statement.

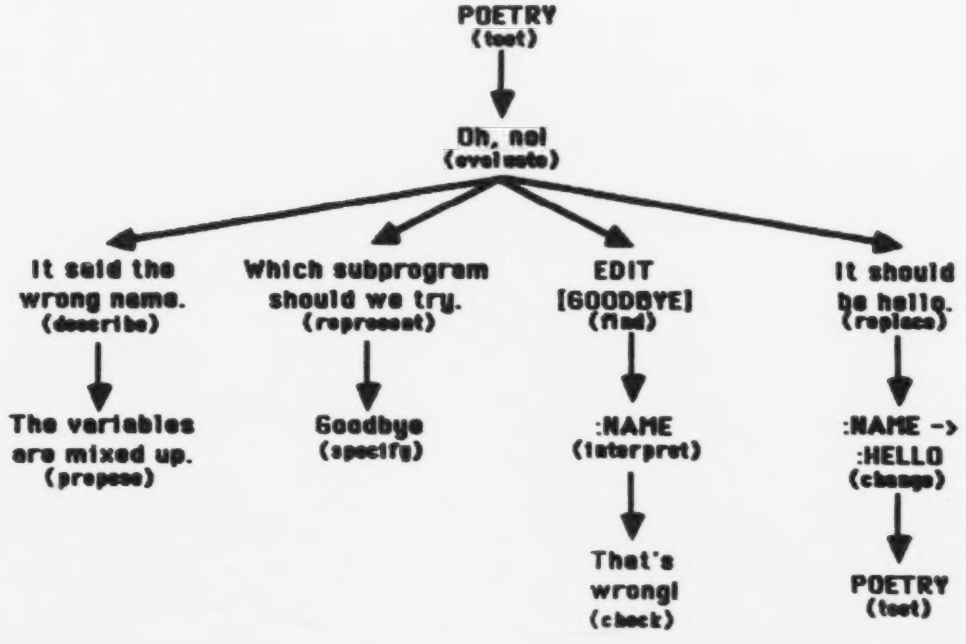


Figure 14: Debugging Success

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests, b) Comparison of the two groups on the list-processing tests, c) Overall result: second mini-course groups found more bugs than first mini-course groups in graphics and list-processing. The maximum number of bugs was six.

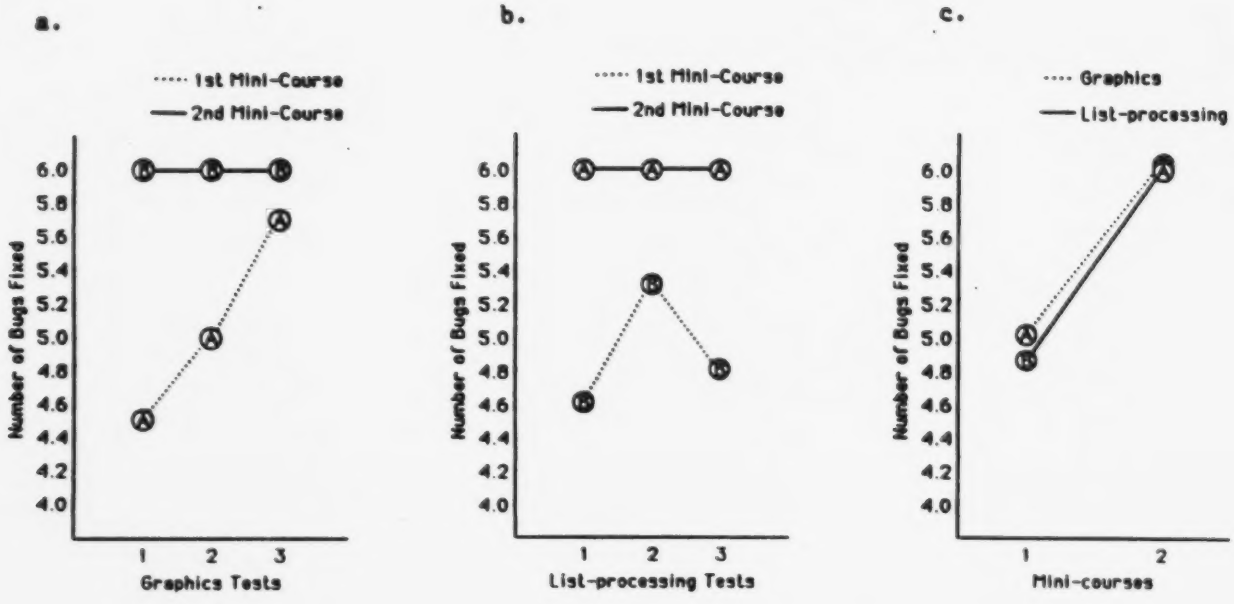


Figure 15: Debugging Speed

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests. b) Comparison of the two groups on the list-processing tests. c) Overall result: second mini-course groups found bugs more quickly than first mini-course groups in graphics and list-processing.

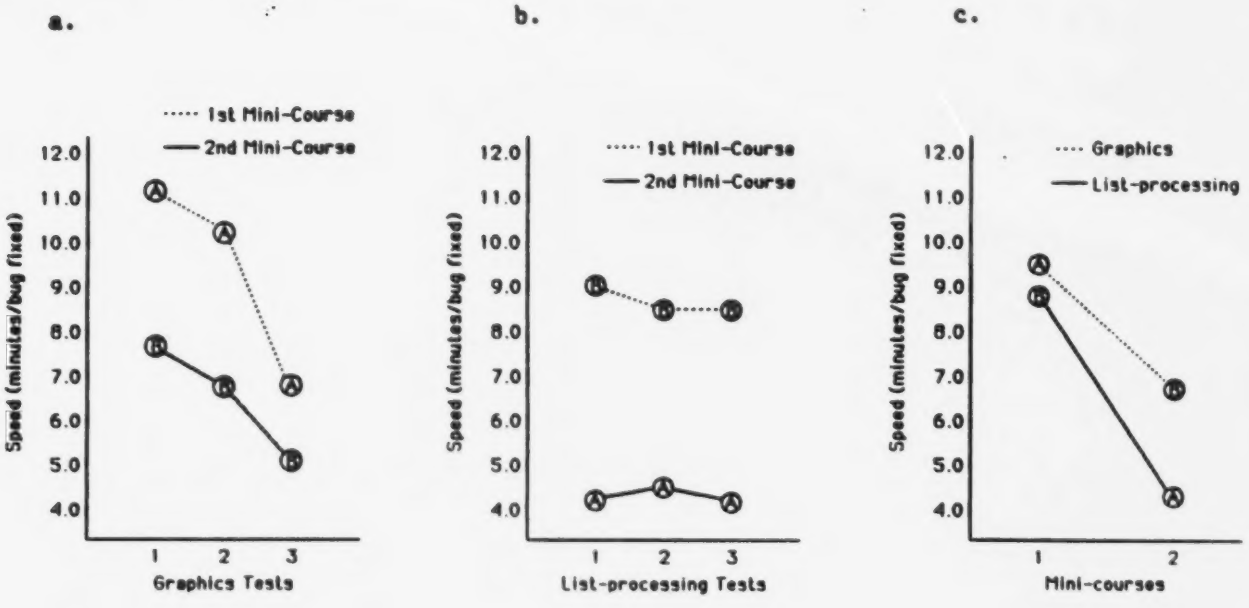


Figure 16: Debugging Efficiency

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests, b) Comparison of the two groups on the list-processing tests, c) Overall result: second mini-course groups took fewer cycles (in terms of the model) to fix bugs than first mini-course groups in graphics and list-processing. All students took fewer cycles on list-processing tests than on graphics tests.

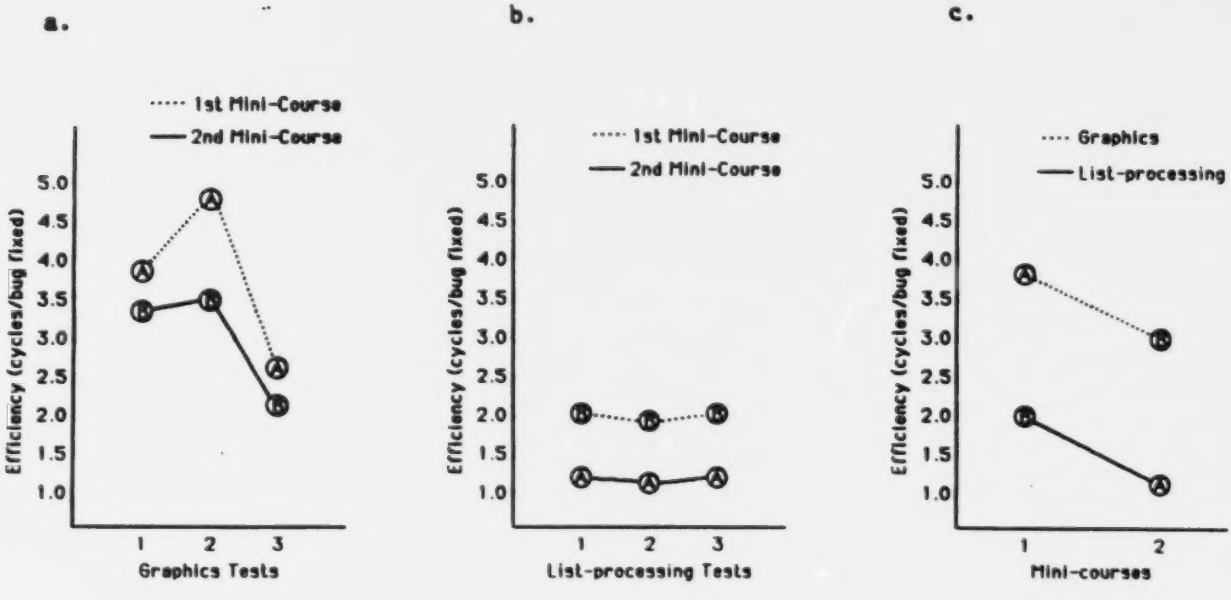


Figure 17: Accuracy of Search Comments

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comments describing the discrepancy, b) Comments proposing the bug. c) Comments specifying the location. Accuracy of search comments was high: however, very few search comments were made. There was no improvement in the accuracy of any type of search comments.

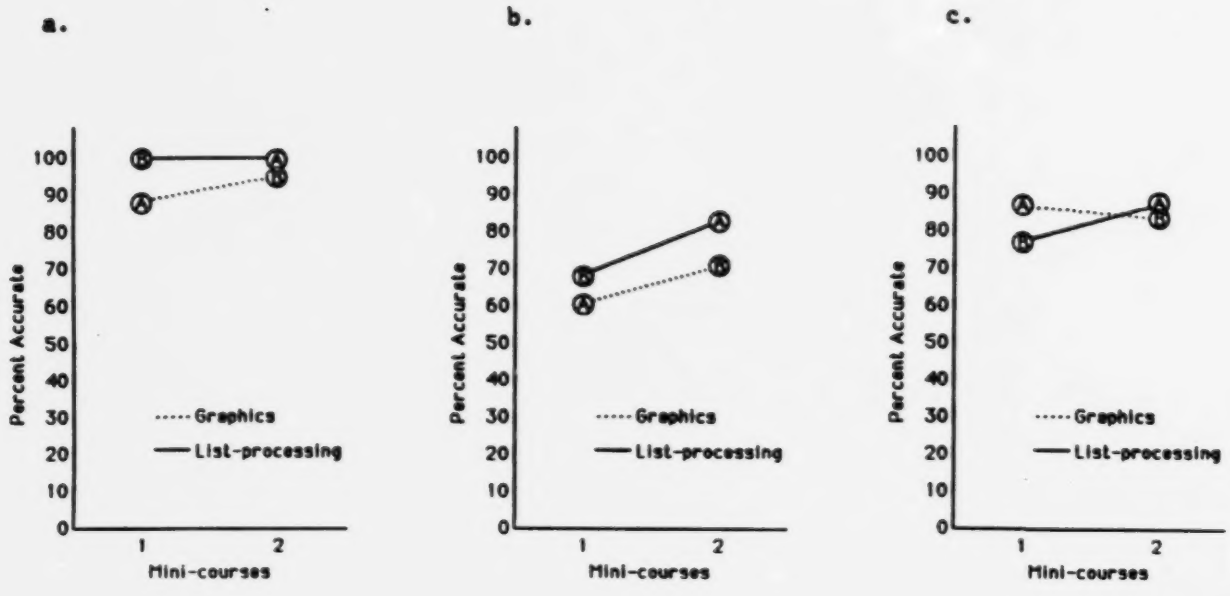


Figure 18: Amount of Pre-Search Comments

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comments describing the discrepancy, b) Comments proposing the bug. c) Comments specifying the location. The percentage of comments made prior to initiating search increased, especially for comments proposing the bug.

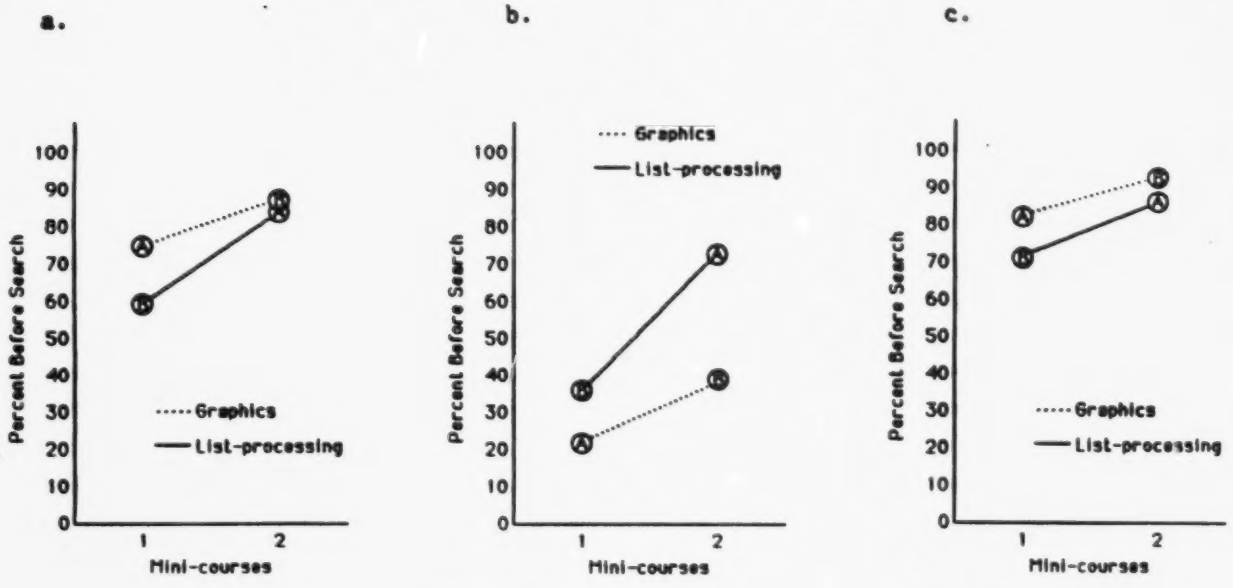


Figure 19: Amount of Program Goal Achieved

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests, b) Comparison of the two groups on the list-processing tests, c) Overall result: there was small improvement in the percentage of program units completed in the second mini-course. Students completed more of the program units on the list-processing tests.

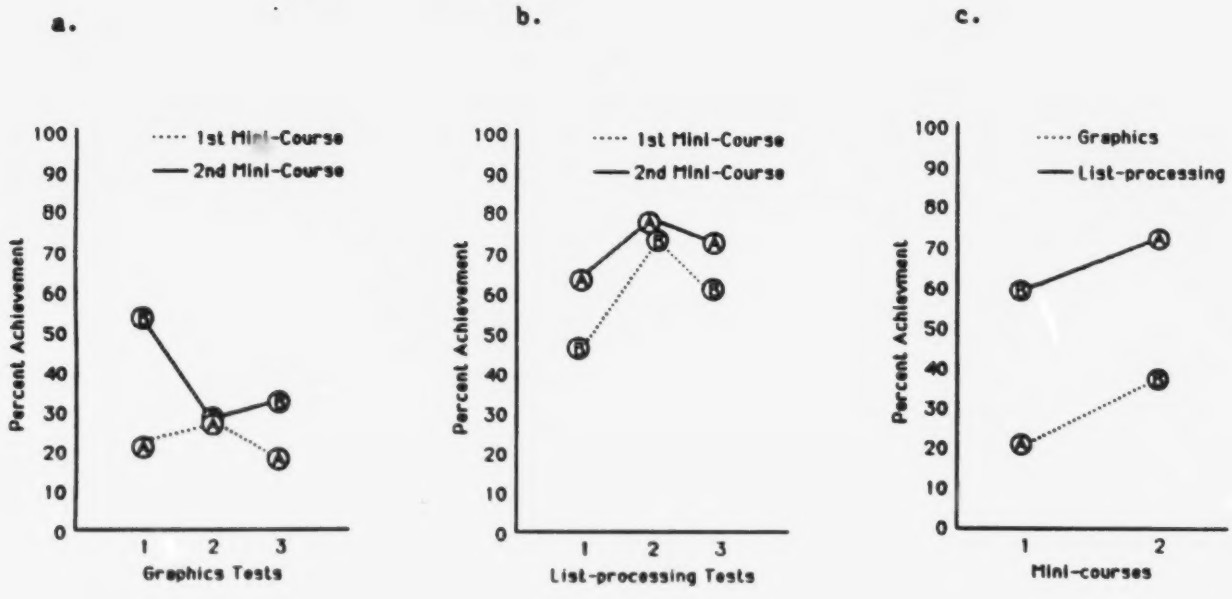


Figure 20: Amount of Program Structure

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests, b) Comparison of the two groups on the list-processing tests, c) Overall result: second mini-course groups added no more structure to their programs than first mini-course groups in graphics and list-processing. Students in list-processing did add more structure than students in graphics, however.

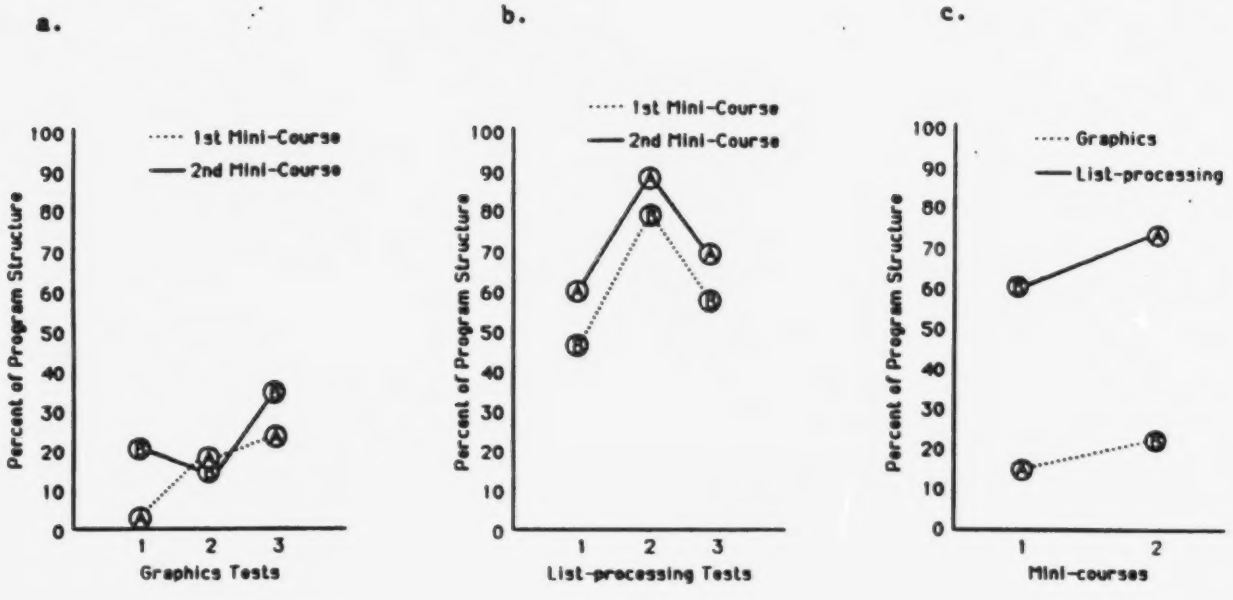


Figure 21: Editing Speed

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests. b) Comparison of the two groups on the list-processing tests. c) Overall result: second mini-course groups were faster at editing than first mini-course groups in graphics and list-processing. Students were faster at editing graphics programs than list-processing programs.

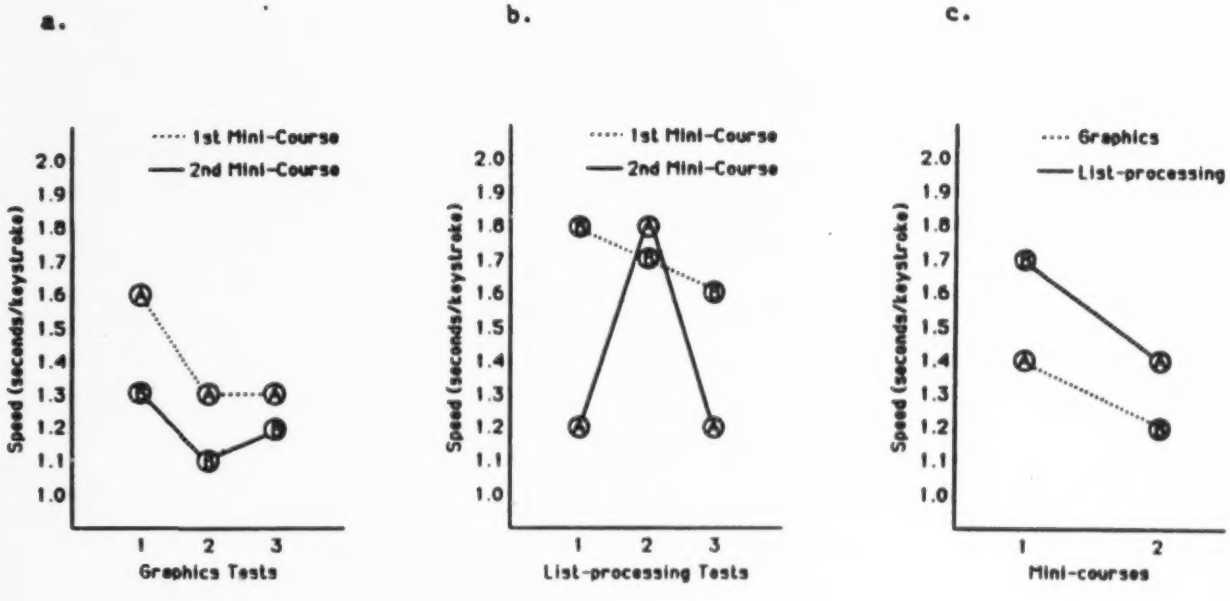


Figure 22: Debugging Speed Minus Editing Speed

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. The connected points show the debugging speed after subtracting the editing speed. For reference, the unconnected points show the debugging speed without adjustment (as in Figure 15). a) Comparison of the two groups on the graphics tests. b) Comparison of the two groups on the list-processing tests. c) Overall result: Subtracting the editing time from the debugging time per bug does not diminish the transfer effect.

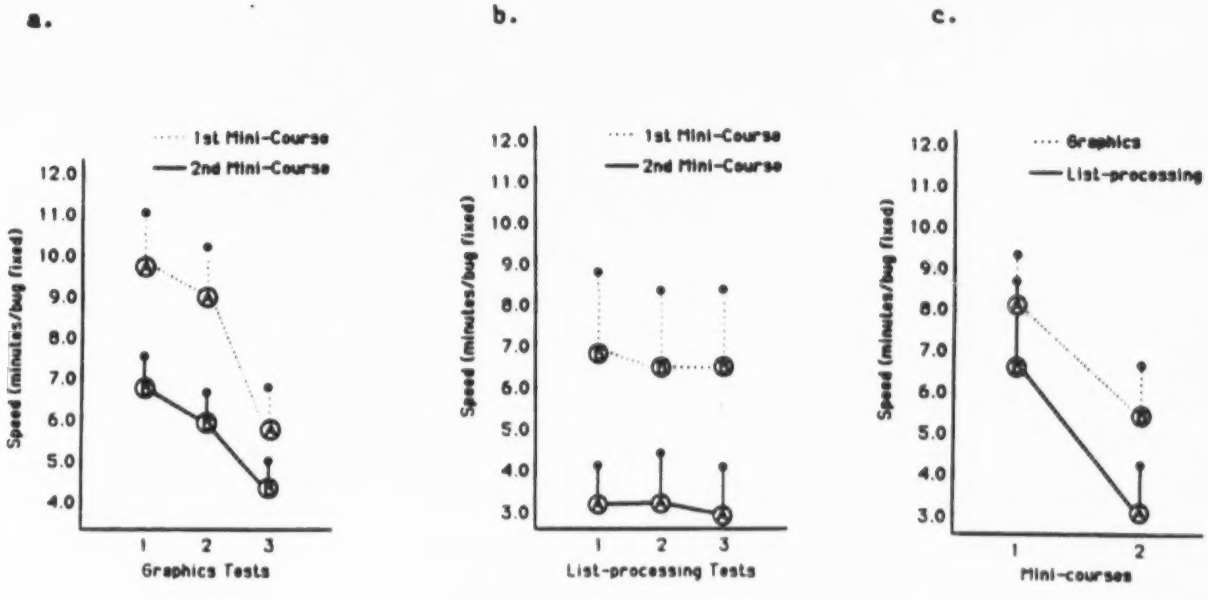


Figure 23: Editing Efficiency

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests, b) Comparison of the two groups on the list-processing tests, c) Overall result: second mini-course groups were more efficient at editing than first mini-course groups in graphics and list-processing.

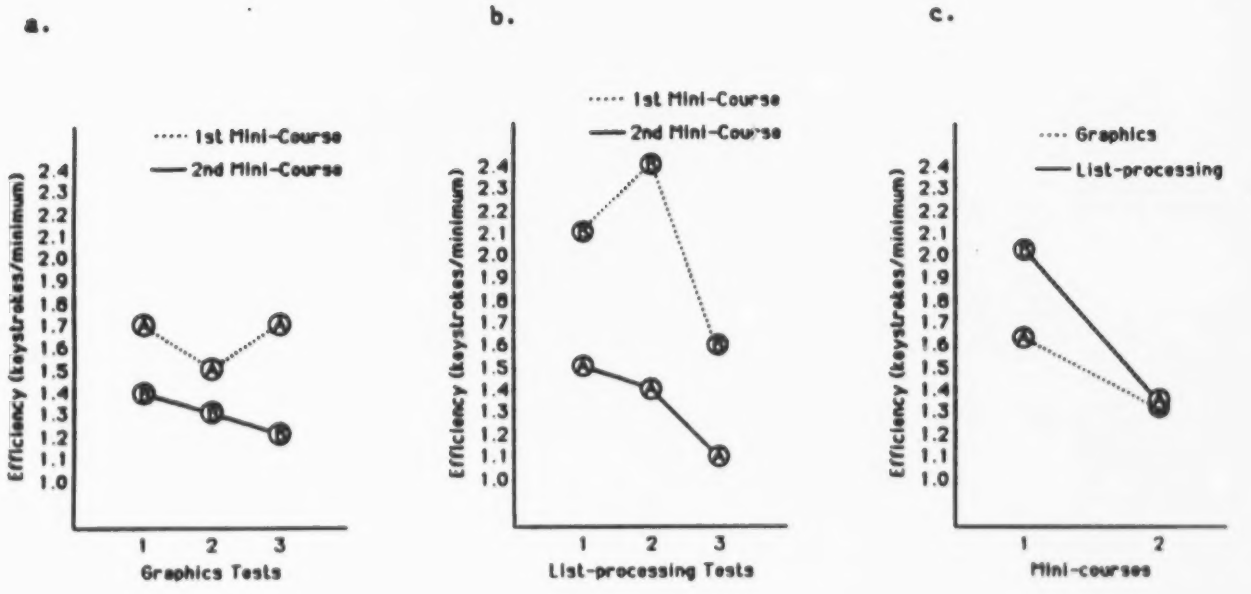


Figure 24: Incorrect Edits

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests. b) Comparison of the two groups on the list-processing tests. c) Overall result: second mini-course groups made fewer incorrect edits than first mini-course groups in graphics and list-processing.

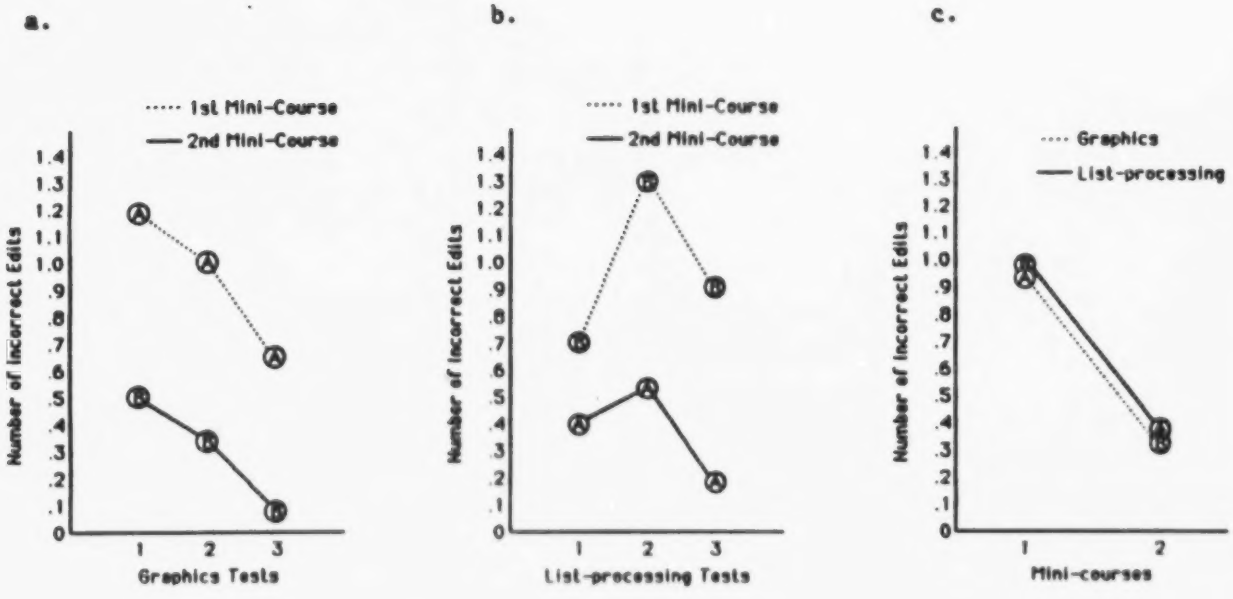


Figure 25: Amount of Help Needed for Editing

Group A took graphics then list-processing; Group B took the mini-courses in the reverse order. a) Comparison of the two groups on the graphics tests. b) Comparison of the two groups on the list-processing tests. c) Overall result: second mini-course groups required less help than first mini-course groups in graphics and list-processing.

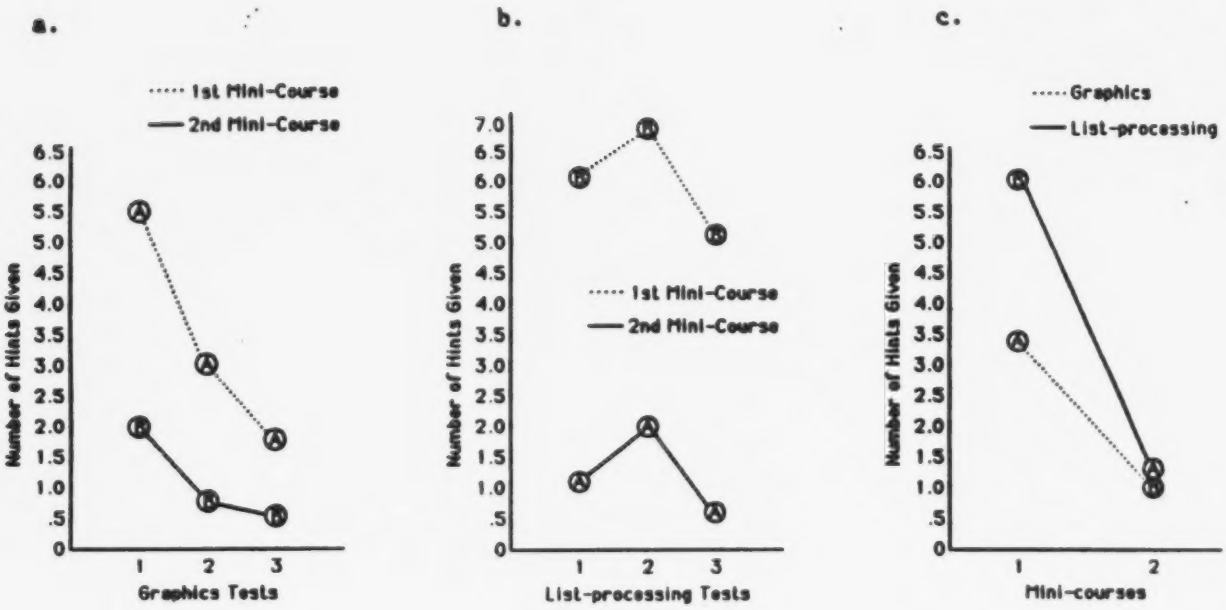
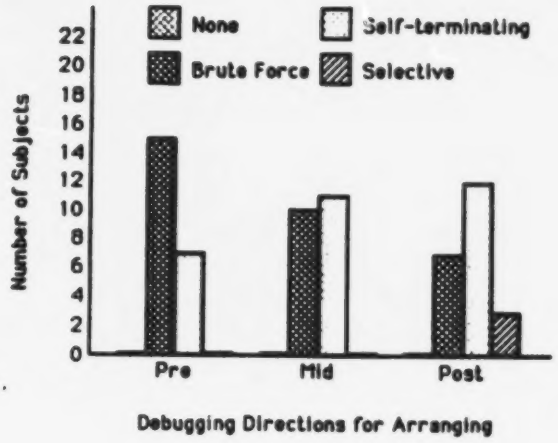


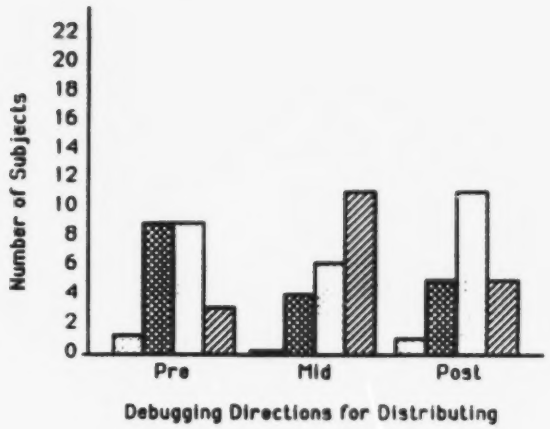
Figure 26: Reading Strategies on Transfer Tests

The number of subjects who used each of the four search strategies for reading the buggy directions. a) On arranging direction tests. b) On distributing direction tests. c) On traveling direction tests. Better strategies are toward the right. Subjects shifted toward better strategies on the mid- and post-tests.

a.



b.



c.

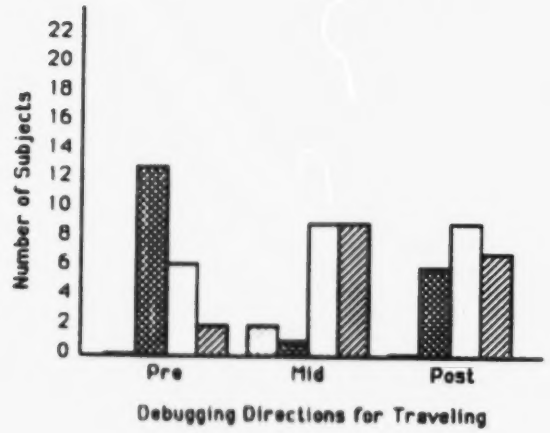
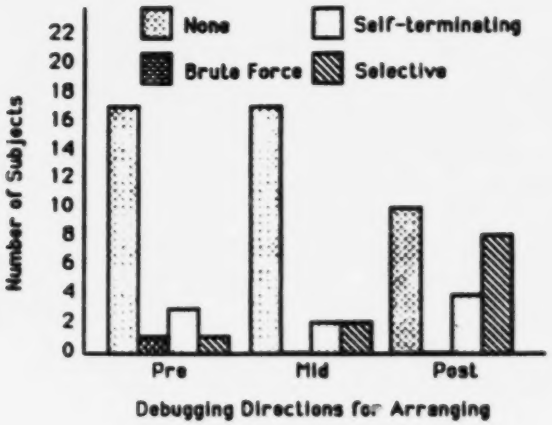


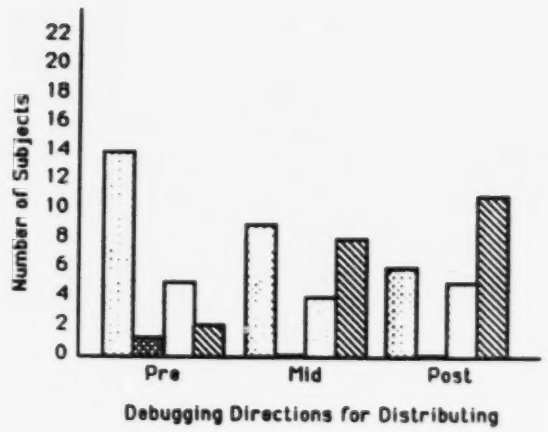
Figure 27: Simulation Strategies on Transfer Tests

The number of subjects who used each of the four search strategies for simulating the buggy directions (checking them against the output). a) On arranging direction tests. b) On distributing direction tests, c) On traveling direction tests. Better strategies are toward the right. Subjects shifted toward better strategies on the mid- and post-tests.

a.



b.



c.

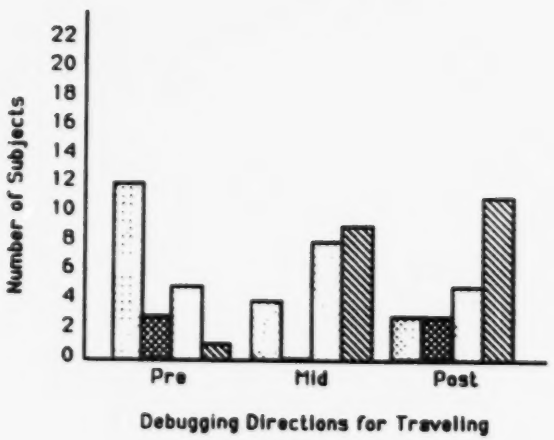


Figure 28: Success for Debugging Directions

On all three types of transfer test, more subjects succeeded in debugging the directions on the mid- and post-tests than on the pre-test.

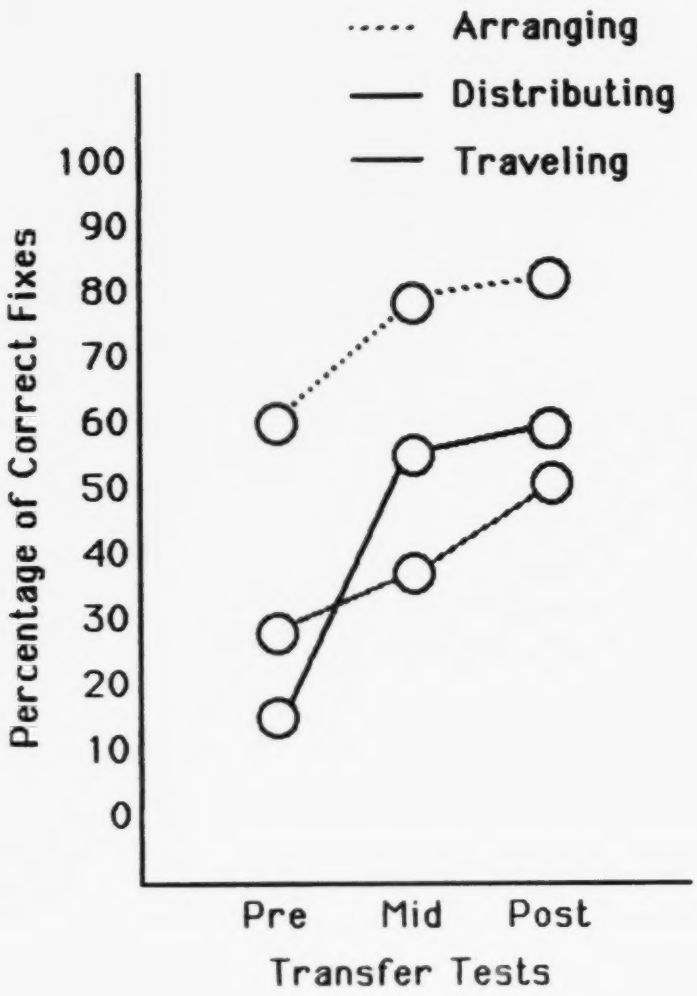


Figure 29: Time to Debug Directions

On all three types of transfer test, subjects took less time to locate the bug on the mid- and post-tests than on the pre-test.

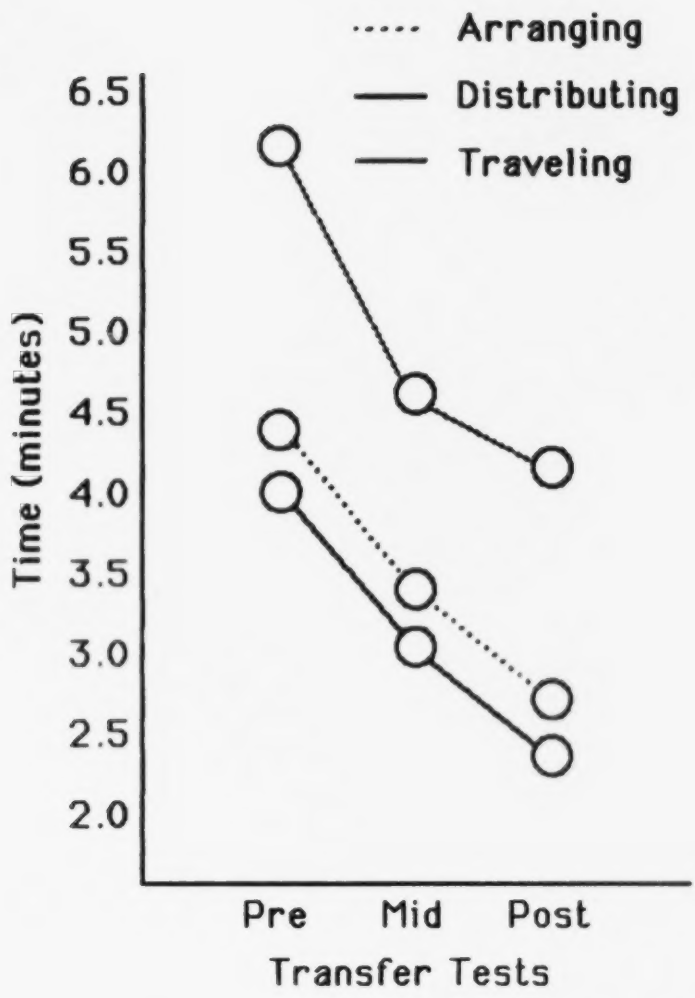
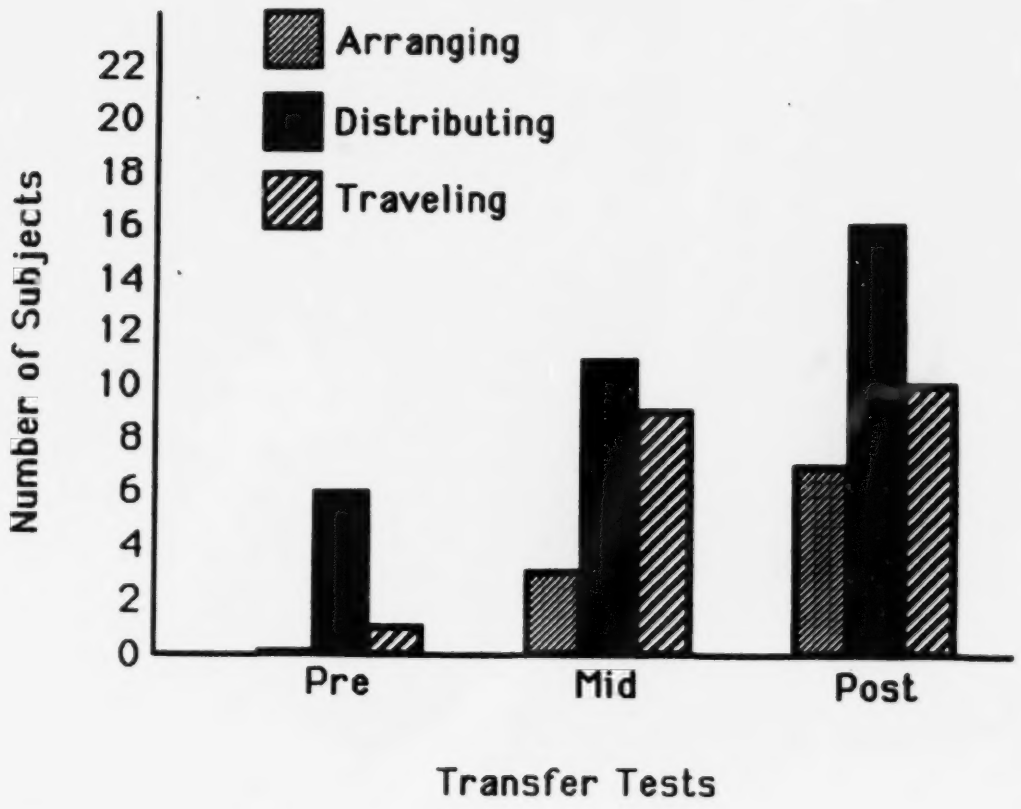


Figure 30: Use of the Checking Strategy

For all three types of transfer test, more subjects checked their fixes (rather than quitting immediately after making a change) on the mid- and post-tests than on the pre-test.



References

Anderson, J.R. (1987). Production systems, learning, and tutoring. In D. Klahr, P. Langley, and R. Neches (Ed.), *Self-modifying production systems: Models of learning and development.* Cambridge, MA: Bradford Books/MIT Press.

Anderson, J.R., & Jeffries, R. (1985, in press). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction*, .

Anderson, J.R., Boyle, C.F., Farrell, R., and Reiser, B.R. (1984). Cognitive principles in the design of computer tutors. In *Proceedings of the Sixth Annual Conference of the Cognitive Science Society.* Boulder, CO: Cognitive Science Society.

Atwood, M.E., & Ramsey, H.R. (1978). *Cognitive structures in the comprehension and memory of computer programs: An investigation of computer debugging* (Tech. Rep. TR-78A21). Army Research Institute for the Behavioral and Social Sciences.

Bassok, M. & Holyoak, K.J. (1986). Schema-based interdomain transfer between isomorphic algebra and physics problems.

Brown, J.S. and Burton, R.B. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, *2*, 155-192.

Brown, S.W., & Rood, M.K. (April 1984). Training gifted students in Logo and BASIC: What is the difference? In *Proceedings of the American Educational Research Association Conference.* New Orleans, Louisiana: AERA.

Bruner, J.S. (1966). On cognitive growth. In J.S. Bruner, R.R. Olver, & P.M. Greenfield (Ed.), *Studies in cognitive growth.* New York: Wiley.

Campbell, P.F., Fein, G.G., Scholnick, E.K., Frank, R.E., & Schwartz, S.S. (April 1985). Initial mastery of the syntax and semantics of logo. In *Proceedings of the American Educational Research Association Conference.* Chicago, Illinois: AERA.

Carver, S.M. and Klahr, D. (1986). Assessing children's logo debugging skills with a formal model. *Journal of Educational Computing Research*, *2*(4), 487-525.

Cheng, P.W., Holyoak, K.J., Nisbett, R.E., & Oliver, L.M. (1986). Pragmatic versus syntactic approaches to training deductive reasoning. *Cognitive Psychology*, Vol. *18*.

Clements, D.H. (1985). Differential effects of computer programming (Logo) and computer-assisted instruction on executive processes and cognitive development. Paper presented at SRCD, Toronto, CANADA, April.

Clements, D.H. & Gullo, D.F. (1984). Effects of computer programming on young children's cognition. *Journal of Educational Psychology*, *76*(6), 1051-1058.

Cole, M., & Griffin, P. (1980). Cultural amplifiers reconsidered. In D.R. Olson (Ed.), *The social foundations of language and thought: Essays in honor of Jerome S. Bruner.* New York: W.W. Norton,

Cuneo, D.O. (1985). Young children and turtle graphics programming: Understanding turtle commands. Paper presented at SRCD, Toronto, CANADA, April.

Dalbey, J., & Linn, M. (April 1984). Spider world: A robot language for learning to program. In *Proceedings of the American Educational Research Association Conference.* New Orleans, LA: AERA,

Degelman, D., Free, J.U., Scarlato, M., Blackburn, J.M., & Golden, T. (1986). Concept learning in preschool children: Effects of a short-term LOGO experience. *Journal of Educational Computing Research*, *2*(2), 199-206.

Dunbar, K. & Klahr, D. (1986). Development of reasoning skills about complex devices. Working Paper.

Ericsson, K.A. & Simon, H.A. (1984). *Protocol analysis: Verbal reports as data.* Cambridge, MA: The MIT Press.

Fisher, C.A. (1986). How do programmers program: Coping with complexity. Working Paper.

Garlick, S. (1984). Computer programming and cognitive outcomes: A classroom evaluation of Logo. Unpublished Honors Dissertation.

Gentner, D. & Gentner, D.R. (1983). Flowing waters or teeming crowds: Mental models of electricity. In D. Gentner & A.L. Stevens (Ed.), *Mental Models*. Hillsdale, N.J.: Erlbaum.

Gick, M.L., & Holyoak, K.J. (1983). Schema induction and analogical transfer. *Cognitive Psychology, 15*, 1-38.

Goody, J. (1977). *The domestication of the savage mind.* New York: Cambridge University Press.

Gorman, H. Jr., & Bourne, L.E. Jr. (1983). Learning to think by learning Logo: Rule learning in third grade computer programmers. *Bulletin of the Psychonomic Society, 21*(3), 165-167.

Greeno, J.G. (1976). Cognitive objectives of instruction: Theory of knowledge for solving problems and answering questions. In D. Klahr (Ed.), *Cognition and Instruction*. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Gugerty, L. & Olson, G.M. (1986). Comprehension differences in debugging by skilled and novice programmers. In E. Soloway & S. Iyengar (Ed.), *Empirical Studies of Programmers*. Norwood, New Jersey: Ablex Publishing Corporation.

Hawkins, J. (April 1983). Learning Logo together: The social context. In *Proceedings of the American Educational Research Association*. Montreal, Canada: AERA,

Hayes, J.R., & Simon, H.A. (1977). Psychological differences among problem isomorphs. In N.J. Castellan, D.B. Pisoni, and G.R. Potts (Ed.), *Cognitive Theory*. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Holyoak, K.J. & Koh, K. (1986). Surface and Structural Similarity in Analogical Transfer. Working Paper.

Jacobson, G. & Jackson, D. (1986). Measurement of two teaching strategies for computer programming instruction. Poster presented at the Empirical Studies of Programmers conference.

Jeffries, R. (March 1982). A comparison of the debugging behavior of expert and novice programmers. In *Proceedings of the American Educational Research Association.* New York, NY: AERA,

Jenkins, E.A., Jr. (1986). An analysis of expert debugging of LOGO programs. Working Paper.

Kessler, C.M. & Anderson, J.R. (1986). A model of novice debugging in LISP. In E. Soloway & S. Iyengar (Ed.), *Empirical Studies of Programmers.* Norwood, New Jersey: Ablex Publishing Corporation.

Kieras, D.E. & Bovair, S. (1985). *The acquisition of procedures from text: A production-system analysis of transfer of training* (Tech. Rep. 16 (TR-85/ONR-16)). University of Michigan.

Kotovsky, K., Hayes, J.R., & Simon, H.A. (1985). Why are some problems hard? Evidence from Tower of Hanoi. *Cognitive Psychology, 17*(2), 248-294.

Kurland, D.M., & Pea, R.D. (1983). Children's mental models of recursive Logo programs. In *Proceedings of the Fifth Annual Cognitive Science Society.* Rochester, NY: Cognitive Science Society,

Linn, M.C., & Fisher, C.W. (December 17 1983). The gap between promise and reality in computer education: Planning a response. In *Making our schools more effective: A conference for California Educators.* San Francisco, CA: ACCCEL,

Littman, D.C., Pinto, J., Letovsky, S., & Soloway, E. (1986). Mental models and software maintenance. In E. Soloway & S. Iyengar (Ed.), *Empirical Studies of Programmers.* Norwood, New Jersey: Ablex Publishing Corporation.

Mawby, R. (April 1984). Determining students' understanding of programming concepts. In *Proceedings of the American Educational Research Association Conference.* New Orleans, LA: AERA,

Mayer, R.E. (1981). The psychology of how novices learn computer programming. *Computing Surveys, 13,* 121-141.

McBride, S.R. (1985). *A cognitive study of children's computer programming* (Tech. Rep. 8502). University of Delaware.

McGilly, C.A., Poulin-Dubois, D., & Shultz, T.R. (1984). The effect of learning LOGO on children's problem-solving skills.

Mohamed, M.A. (1985). *The effects of learning LOGO computer language upon the higher cognitive processes and the analytic/global cognitive styles of elementary school students.* Doctoral dissertation, School of Education, University of Pittsburgh, Unpublished doctoral dissertation.

Newell, A. & Simon, H.A. (1972). *Human problem solving.* Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Nisbett, R.E., Krantz, D.H., Jepson, C., & Kunda, Z. (1983). The use of statistical heuristics in everyday inductive reasoning. *Psychological Review, 90,* 339-363.

Olson, D.R. (1976). Culture, technology and intellect. In L.B. Resnick (Ed.), *The nature of intelligence.* Hillsdale, NJ: Lawrence Erlbaum Associates,

Ong, W.J. (1982). *Orality and literacy: The technologizing of the word.* New York: Methuen.

Papert, S. (1972). Teaching children thinking. *Programmed Learning and Educational Technology, 9,* 245-255.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* New York: Basic Books.

Papert, S. (April, 1986). The next step: LogoWriter. *Classroom Computer Learning.* , pp. 38-40.

Papert, S., Watt, D., DiSessa, A., & Weir, S. (1979). *Final report of the Brookline LOGO Project: An assessment and documentation of children's computer laboratory.* Cambridge, MA: MIT Division for Study and Research in Education.

Pea, R.D. (April 1983). Logo programming and problem solving. In *Proceedings of the*

American Educational Research Association Conference. Montreal, Canada: AERA, Also Tech Report number 12.

Pea, R.D., & Kurland, D.M. (1984). On the cognitive effects of learning computer programming. In *New Ideas in Psychology.* Elmsford, NY: Pergammon.

Pea, R.D., Hawkins, J., & Sheingold, K. (April 1983). Developmental studies on learning Logo computer programming. In *Proceedings of the Society for Research in Child Development Conference.* Detroit, MI: SRCD, Also Tech Report number 17.

Perkins, D.N. & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Ed.), *Empirical Studies of Programmers.* Norwood, New Jersey: Ablex Publishing Corporation.

Reed, S.K., Ernst, G.W., & Banjeri, R. (1974). The role of analogy in transfer between similar problem states. *Cognitive Psychology, 6,* 436-450.

Roberts, R.J. Jr. (May 1984). *Young children's spatial frames of reference in simple computer graphics programming.* Doctoral dissertation, Department of Psychology, University of Virginia,

Sauers, R. & Farrell, R. (1982). GRAPES User's Manual. Department of Psychology, Carnegie-Mellon University.

Schwartz, T.A., Evans, H., & Caritj, W.H. (April 1984). Looking into a large-scale Logo project. In *Proceedings of the American Educational Research Association Conference.* New Orleans, LA: AERA,

Singley, M.K. and Anderson J.R. (1985). The transfer of text-editing skill. *International Journal of Man-Machine Studies, 22,* 000-000.

Smith, S.B. (1986). *How what we learn effects transfer: An identical elements evaluation of transfer between isomorphic problems.* Doctoral dissertation, Department of Psychology, Carnegie-Mellon University, Unpublished doctoral dissertation.

Spohrer, J.C. & Soloway, E. (1986). Analyzing the high frequency bugs in novice programs.

In E. Soloway & S. Iyengar (Ed.), *Empirical Studies of Programmers*. Norwood, New Jersey: Ablex Publishing Corporation.

Spohrer, J.C., Soloway, E., & Pope, E. (April 1985). Where the bugs are. In *CHI '85 Proceedings*. ,

Thorndike, E. (1913). *Educational psychology, Vol. II, The psychology of learning*. New York: Teachers College, Columbia University.

Vygotsky, L.S. (1978). *Mind in society*. Cambridge, MA: Harvard University Press. M. Cole, V. John-Steiner, S. Scribner, & E. Souberman [Eds.].

Webb, N.M. (April 1984). Problem-solving strategies and group processes in small groups learning computer programming. In *Proceedings of the American Educational Research Association Conference*. New Orleans, LA: AERA,

Winston, P.H. (1977). *Artificial intelligence*. Reading, MA: Addison-Wesley Publishing Company.

Yant, S.E. (1986). The effect of LOGO learning on children's planning skills. Senior Honors Thesis, Carnegie-Mellon University.